

NX Check-Mate Author Training

Reference Material

Proprietary and restricted rights notice

This software and related documentation are proprietary to Siemens Product Lifecycle Management Software Inc.

© 2013 Siemens Product Lifecycle Management Software Inc. All Rights Reserved.

All trademarks belong to their respective holders.

Contents

Proprietary and restricted rights notice	3
Contents	5
Course Overview	9
Preface	9
Introduction	9
Course Description	11
Intended Audience	11
Course Objectives	11
Details	11
Prerequisite Knowledge	12
How to use this manual	12
Lesson format	12
Activity format	12
Learning tips	12
Section 1: Environment Variables	13
Storing Customized Checks and Profiles	13
Storing Check-Mate Results (log files)	13
Configuring the Display of Checks Available in NX	14
Pre-selection of Checks for Execution	14
Automatic Execution of Checks or Profiles	15
User or Group-specific Configuration	15
A Few Other Environment Variables	15
NXCustom	16
Activity 1 - Defining The Check-Mate Environment	16
Activity 2 – Using Individual Checks Out-of-the-box	17
Activity 3 – Using a Check-Mate Profile	21
Deploying Custom Checks into an NX Session	25
Activity 4 – Configuring Checker Visibility	25
Activity 5 – Pre-populating the Chosen Tests Area	26
Activity 6 – Invoking the Profile Automatically	27
Section 2: Customer Defaults	31
General	31
HD3D	32
Run Options	33
Heal	34
Author Tests	35

Section 3: Advanced Check-Mate Customization	36
The Check-Mate Configuration Continuum.....	36
Activity 7 – Creating a Check-Mate Profile.....	37
Activity 8 – Using a Template Check.....	44
Activity 9 – Modifying an Existing Check to Create a New Check	50
Creating New Checks from Scratch	55
Section 4: Knowledge Fusion	59
Overview.....	59
General Qualities	59
Programming elements	60
Activity 10 - How to Add Attributes	61
Activity 11 - Defining Attributes Using Expressions	70
Activity 12 - Loops.....	71
Activity 13 - Functions	83
Section 5: Check-Mate Specific KF Elements.....	91
Anatomy of a Check-Mate Check	91
Anatomy of a Check-Mate Profile.....	96
Checker Customized Dialog Details.....	99
Logging Results.....	100
Interactive log files	101
Batch log files.....	101
Logging Checker Tags.....	101
Special Attributes for Checks.....	102
Activity 14 - Understanding the Check-Mate dfa.....	103
General Tips for creating Checks	106
Finding the Functions You Need.....	107
Customizing Check-Mate Feedback in HD3D.....	109
Section 6: Checker Debugging and Troubleshooting	111
Special Text Editors.....	111
Knowledge Fusion Navigator.....	111
Reload All	112
Add Attribute	112
Add Child Rule.....	112
NX Log File	113
Using Debugging Print Statements.....	113
Code Organization.....	114
Performing Calculations Inside do_check	114
Performing Calculations Outside do_check.....	115
Activity 15 - Check for syntax errors	116
Activity 16 - Adding debug print statements	116
Activity 17 - Add a check as a child rule.	117
Section 7: Tips for Successfully Implementing Check-Mate.....	119

Overview	119
Environment and File Locations	119
Simplifying Check-Mate for Improved Efficiency.....	120
Simple Organization	121
Using Variables to Simplify the Interface.....	121
Complete Automation.....	121
Section 8: Check-Mate and Teamcenter	123
Storing Check-Mate Results to Teamcenter.....	123
Why save immediately after checking?	123
What if I can't see the Validation Results Summary?	125
Section 9: Check-Mate External Viewers	126
Activity 18 – Check-Mate External Viewers	126

Course Overview

Preface

If there's any doubt that lack of quality costs industry a ton of money, get this statistic:

"As much as one out of four factory workers produce nothing at all. They spend their entire work day fixing the mistakes of other workers."

-- Lee Iacocca Former Chairman of Chrysler Corp.

Introduction

Check-Mate is a powerful tool for checking the quality of NX models. A “check” is a small piece of logic that looks for a particular condition within a model. Individual checks may validate anything from layering conventions to following company drafting standards to various modeling best practices to verifying correct usage of reference sets and WAVE links and techniques for organizing and working with assemblies. All in all, Check-Mate is a great tool for verifying that various things in a model have been “done the right way”.

As installed by default during the standard NX installation, Check-Mate is preconfigured to immediately start providing benefit to users. Roughly 300 checks are immediately available and ready-to-use without any additional configuration effort.

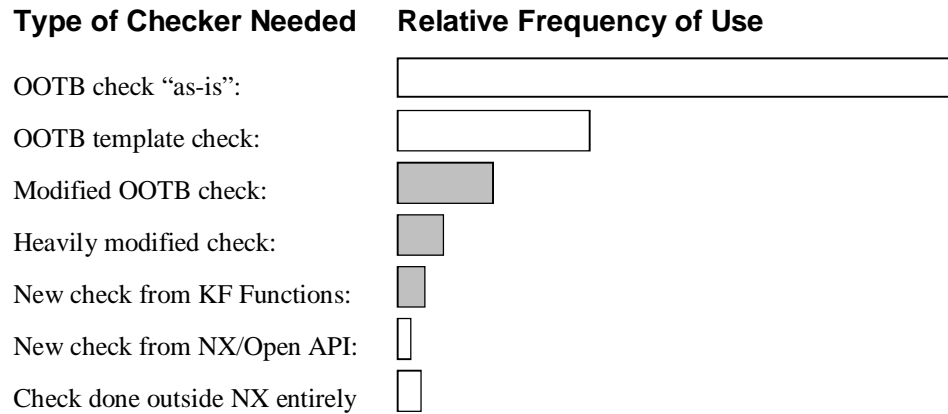
In many cases, however, companies will want users to consistently be validating their designs using a predetermined set of company-specific validations. NX has provided a variety of tools for making this process easier for the users. One such tool is the concept of a “profile”.

A Check-Mate “profile” is a collection of checks that will be executed together at the same time. Checks contained in a profile can be pre-configured with any default values or needed input parameters. A profile is a great tool for ensuring that a complete set of checks is performed using a desired set of quality criteria. It is also a great tool for streamlining the user experience, as it removes the requirement for users to manually collect, configure, and run individual checks.

To go a step further, NX also provides an optional set of tools for making it harder for users to AVOID validation of their designs. Companies can choose the specific level of enforcement they would like to employ with respect to Check-Mate validation.

The Check-Mate Author Tools dialog (on the Check-Mate toolbar) lets you create/edit profiles, create/edit checkers, or configure a checker using the Knowledge Fusion Language (KF) to define rules, specify standards, and determine executions.

There are several levels to configuring and customizing Check-Mate.



This course will concentrate on the highlighted items

Course Description

Check-Mate Author is an advanced method based course that addresses the NX product validation functionality “Check-Mate”. This advanced course is intended to introduce the process of creating custom Check-Mate profiles and checkers. The course will cover the basics of Knowledge Fusion (KF) syntax (Check-Mate checkers are written in the KF language) as well as how to write and use custom Check-Mate checkers.

Intended Audience

This material is suited for CAx system administrators, CAx Group Managers, NX Super-users, and others who may be called upon to configure and/or deploy Check-Mate across groups of users.

Parts of this material are also intended to aid those who will programmatically edit or create new custom Check-Mate checkers or profiles. This latter portion of this material will require a basic understanding of Knowledge Fusion programming.

Course Objectives

After successfully completing this course, you should be able to perform the following activities in NX:

- Create and manage custom Check-Mate profiles and checkers.
- Configure the Check-Mate environment for effective use.

Details

This course covers general procedures on how to create and edit profiles and checkers interactively as well as directly in DFA files (KF code). It also provides information concerning Check-Mate configuration and implementation. Topics include:

- Environment variables for configuring Check-Mate.
- Configuring, modifying checker and profiles using interactive dialogs in NX Check-Mate.
- Detailed information about Knowledge Fusion (KF) code syntax and structure.
- Using KF to customize and extend the functionality of Check-Mate.
- Best practices for writing and debugging Check-Mate KF code
- Implementing Check-Mate on site.

Prerequisite Knowledge

- Basic understanding of how to change NX customer defaults and set system environment variables.
- Working knowledge of the NX Interface.
- Basic understanding of using Check-Mate to test parts. Basic knowledge of programming practices (variables, structure, logic, loops, etc.). Knowledge Fusion programming experience is a plus.

How to use this manual

The following guidelines describe how you can get the most benefit from your use of the course guide and the accompanying activities.

Lesson format

The general format for lesson content is:

- Instructor presentation
- One or more activities
- Summary

Activity format

Activities have the following format:

1. This is an example of a step. Numbered steps specify the actions you will perform.
 - a. Sub-steps detail how to complete the step.



Always read the Cue and Status information while working through activities and as you perform your regular duties.



As you gain skills you may need only to read the step text to complete the step.

Learning tips

- Ask questions.
- Confirm important facts by restating them in your own words.

Section 1: Environment Variables

Storing Customized Checks and Profiles

If a company decides to pre-configure their Check-Mate environment to deliver company-specific checks and/or profiles to their users, a very early consideration must be the location in which customized Check-Mate content will be stored.

NX has the ability to find Check-Mate content in a variety of places, depending on a company's deployment preferences. NX will actually automatically look in several places for checks – these various locations are detailed in the Check-Mate documentation in the Environment Variables section. For the purpose of today's class, however, we will introduce the method that NX “looks for” first as it searches for Check-Mate content.

After the pre-installed Check-Mate classes, the first place NX will look for Check-Mate content is in the folder referenced by the environment variable **UGCHECKMATE_USER_DIR**. NX will recursively search for any checks or profiles it can find in the folder referenced by this environment variable (or any of its subfolders).

In a real deployment, you will likely want to put this folder on a shared network drive so that everyone is always looking at the same definition of the checks and profiles. This will also keep the checking profiles isolated from your NX installation and allow you to change them in one place, rather than on each individual computer. You will likely also want to write-protect the folder containing the actual check and profile code, so that the standards stay “standard”.

Commonly, customers will use a batch file to start NX that contains this environment variable (and a few others we'll discuss in this section). Depending on the way you deploy NX at your company, you might choose instead to set this environment variable using the Windows Control Panel, or choose to set this environment variable in the `ugii_env.dat` file. Any of these will work equally well.

Storing Check-Mate Results (log files)

If you are using NX in conjunction with Teamcenter, then you may want to store Check-Mate results directly into the Teamcenter database. This process will be described in a later section of this document.

If you are using NX in native mode (without Teamcenter) or if you would like to perform supplemental report generation using the XML-based Check-Mate log files, then you may want to collect Check-Mate log files into one or more central locations.

The environment variable **UGII_CHECKMATE_LOG_DIR** provides a default destination for log files created by Check-Mate. Setting this value consistently across a set of users can result in a useful body of log files being deposited in a central location over time. This can provide valuable insight into the quality status of a large project, and in some cases can help uncover areas in which users may require reminders or training regarding certain standards.

Configuring the Display of Checks Available in NX

While a large number of checks are available OOTB with NX, the vast majority of customers really only want their users to be using some subset of that list. Often, a company may only want their users to be running one particular company standard profile, for example. There are four environment variables that can be used to configuring the checks that are visible in the Check-Mate *Set Up Tests* dialog inside NX. These environment variables are:

UGCHECKMATE_ALLOW_CATEGORY
UGCHECKMATE_HIDE_CATEGORY
UGCHECKMATE_ALLOW_CHECKER
UGCHECKMATE_HIDE_CHECKER

These four variables can work in coordinated ways to significantly reduce the complexity of the interface presented to the end user. This reduces confusion on their part by removing unnecessary options and complexity. Single values can be assigned to these variables in a batch file using a syntax like this:

set UGCHECKMATE_ALLOW_CATEGORY=TBA_Aero

Multiple checkers or profiles or categories can be specified by separating the values with commas:

set UGCHECKMATE_ALLOW_CATEGORY=TBA_Aero,TBA_Checks

It is important to note that while these variables control the visibility of checks, they do not control the availability of checks for use within visible profiles. For example, if a company chose to make all individual checks invisible, these checks would still be available for use within the context of a visible profile.

Pre-selection of Checks for Execution

Taking the previous tool a step further, yet another environment variable can be used to pre-populate the “Chosen Tests” window in the Check-Mate user dialog.

UGII_CHECKMATE_DEFAULT_CHECKER can be used to identify one or more checkers or profiles that should be added to the “Chosen Tests” list by default. Again, this option can further reduce confusion and complexity to streamline the user experience. Like the visibility controls above, a default checker or profile can be assigned to these variables in a batch file using the programmatic class name (as opposed to the display name) and a syntax like this:

set UGII_CHECKMATE_DEFAULT_CHECKER=TBA_Profile

Similarly, multiple checkers or profiles can be specified by separating the values with commas:

set UGII_CHECKMATE_DEFAULT_CHECKER=TBA_Profile,Mfg_Profile

Automatic Execution of Checks or Profiles

Used in conjunction with the previous option, the environment variable **UGII_CHECKMATE_ALLOW_AUTO_RUN** can “turn on” the ability to have a check or profile automatically executed every time parts are saved, for example. Effective use of this option can ensure that users are consistently validating their designs, and even prevent users from avoiding validation of their parts. This option also requires proper configuration of a profile that will be automatically executed, by setting the `check_time_index:` attribute in the profile as follows:

check_time_index: attribute value:	Effect on Profile Execution:
1	Profile will only be executed Manually
2	Profile will be automatically executed at each NX update
3	Profile will be automatically executed at each NX save operation

User or Group-specific Configuration

Note that if different users in different departments or different design groups are presented with a different set of values for the above environment variables, their entire Check-Mate environment can be very specifically controlled.

This may be accomplished either by providing different work groups with a unique startup batch script, or by pre-setting certain environment variables at the system level for certain users or groups.

A Few Other Environment Variables

There are actually a few more environment variables for further refining the user experience, but we will not introduce them in this short course. These variables are described in detail in the Environment Variables section of the Check-Mate chapter in the standard NX documentation.

NXCustom

Use NXCustom to easily define and manage the Check-Mate environment variables.

The NXCustom template will also be used to locate the Check-Mate custom profiles and checks which will be used, or created, during the training.

Activity 1 - Defining The Check-Mate Environment

In this activity you will use NXCustom to help define the Check-Mate environment. You will use NXCustom throughout the remaining activities to launch NX. This activity must be completed successfully before any other activities can be accomplished.

Step 1 : Install NXCustom to the Desktop.

Step 2 : Create an NXCustom library directory called 'Checkmate'.

(Note this is not the correct spelling of Check-Mate)

Step 3 : Copy the preloaded profile 'cm_train_preloaded_profile' to the 'Checkmate' directory.

Step 4 : Add the environment variables, in 'CM_Training_config.dat', to the NXCustom '*env.dat'.

[END of Activity]

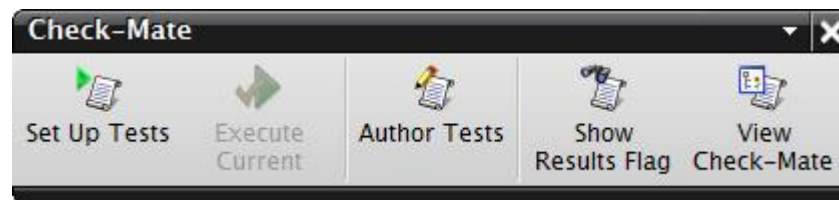
Activity 2 – Using Individual Checks Out-of-the-box

In the first activity we will review the OOTB behavior and interaction with Check-Mate by collecting a few out-of-the-box checks and running them against a part. This will show the way Check-Mate can be used without any configuration whatsoever.

Step 1: Start NX and open the part Check-Mate_part_01.prt.

Step 2: Start the Check-Mate application

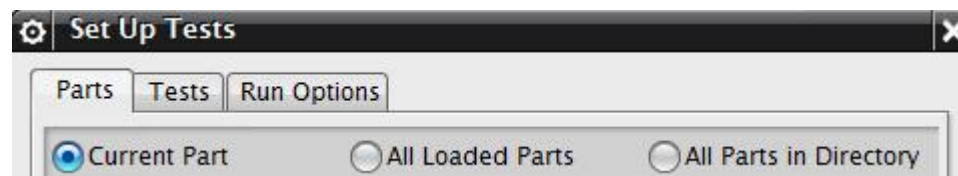
- ☐ Turn on the Check-Mate toolbar by right-clicking in the toolbar area and choosing **Check-Mate** from the list. (It should be quite near the top of the list.) Or Select Check-Mate from HD3D Tool Bar.
- ☐ Choose **Run Tests** from the Check-Mate toolbar (the first icon shown here, with the green arrow on it.)



- ☐ Alternately, choose Analysis → Check-Mate → Run Tests in the pull-down menus.

Step 3: Choose the parts against which to run Check-Mate tests.

- ☐ Make sure the “**Parts**” tab is selected, as shown here.

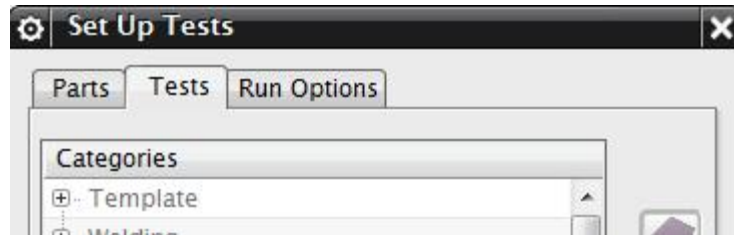


- ☐ Select “*Current Part*” at the top of this tab to run this set of checks against the current part.

(NOTE: The second option, “*All Loaded Parts*” is a good way to check all of the parts in a simple assembly – load the entire assembly, and choose this option. The third option, “*All Parts in Directory*” is a great way to check the results of a translation or verify a set of data received from a supplier or OEM. The folder specified in the third option can be either an O/S folder or a folder in Teamcenter.)

Step 4: Choose the tests that will be run.

- ☐ Proceed to the second tab, called “**Tests**” by clicking on this tab.

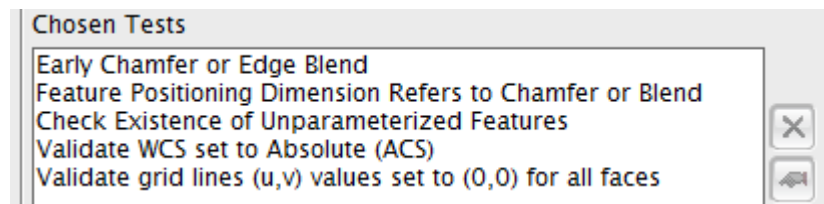


- ☐ From the tree structure, find and select the following set of checks, and using the green down arrow, add them to the “*Chosen Tests*” area below. (A hint for finding each check is given in parenthesis after the name of each check.)

Early Chamfer or Edge Blend (*Modeling.Features*)
Feature Positioning Dimension Refers to Chamfer or Blend (*Modeling.Features*)
Check Existence of Unparameterized Features (*Modeling.Features*)
Validate WCS set to Absolute (*File Structure*)
Validate grid lines (u,v) values set to (0,0) for all faces (*File Structure*)

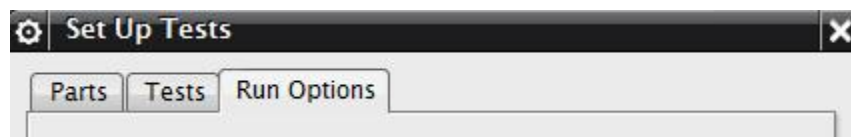
(NOTE: When you have some free time, poke around in this list. You’ll see that there is a huge variety of checks available here for not just modeling, but drawings, assemblies, routing, and other things as well.)

- ☐ When you are done, the lower list should look like this:



Step 5: Run the selected tests against the selected part(s).

- ☐ Once your list is complete, proceed to the third tab, “**Run Tests**”.

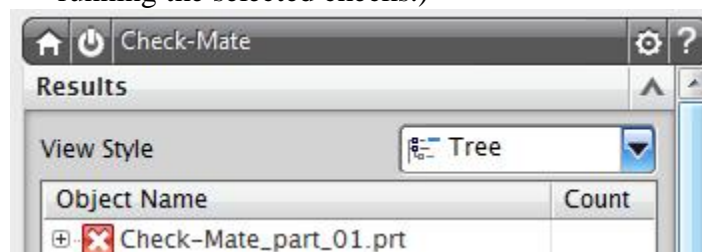


- ❑ Leave all selections at their default state for now. Near the bottom, press the “**Execute Check-Mate**” button to run the selected checks against the selected parts.

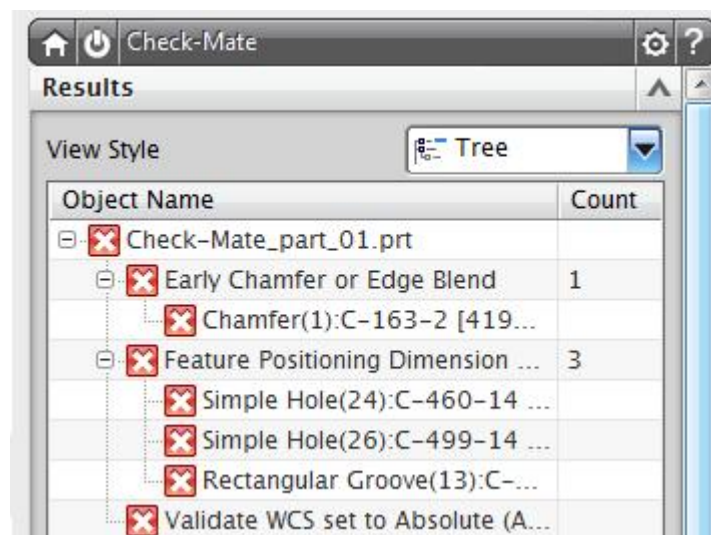


Step 6: Examine the results.

- ❑ Once the checks have completed running close the dialog, the Check-Mate results will be displayed in the results section of the HD3D Tools Check-Mate gadget. (You might have noticed the status line telling you that NX was running the selected checks.)



- ❑ The tree here will describe the errors found in this part. Expand the tree as shown to find the three errors.



- What are the three errors?
- What more information about these errors can you find in the tree?

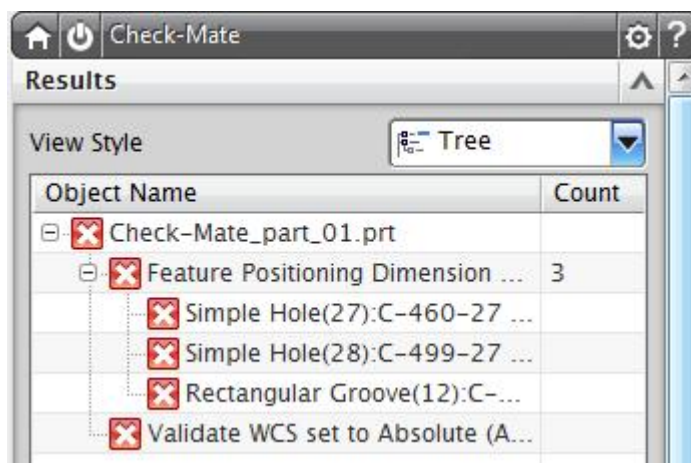
(NOTE: RMB on the various items in the tree gives options to examine and do more in the graphics window.)

Step 7: Fix the *Early Chamfer or Edge Blend* error.

- ☐ Open the Part Navigator.
- ☐ Confirm that you are in the Modeling application.
- ☐ Drag the early chamfer to the bottom of the model history.

Step 8: Confirm that this error has been fixed by running Check-Mate again.

- ☐ Go to HD3D Tools tab
- ☐ In the controls tab press the “**Execute Check-Mate**” button again.
- ☐ Observe in the “View Results” tab that the *Chamfer/Blend* check now passes, it is no longer present in the list.



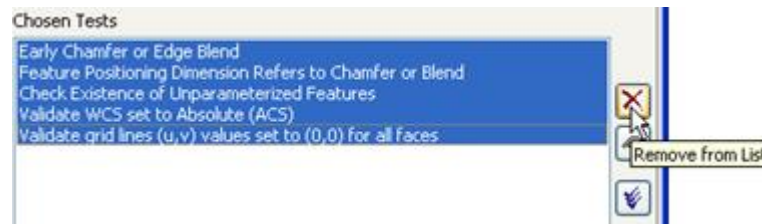
[End of Activity]

Activity 3 – Using a Check-Mate Profile

In this second activity we will run the same five checks but do so using a Check-Mate **Profile**. This will show a more efficient way for your users to interact with the standard set of checks that they should be performing.

Step 1: Remove the individual checks from the list of Chosen Tests.

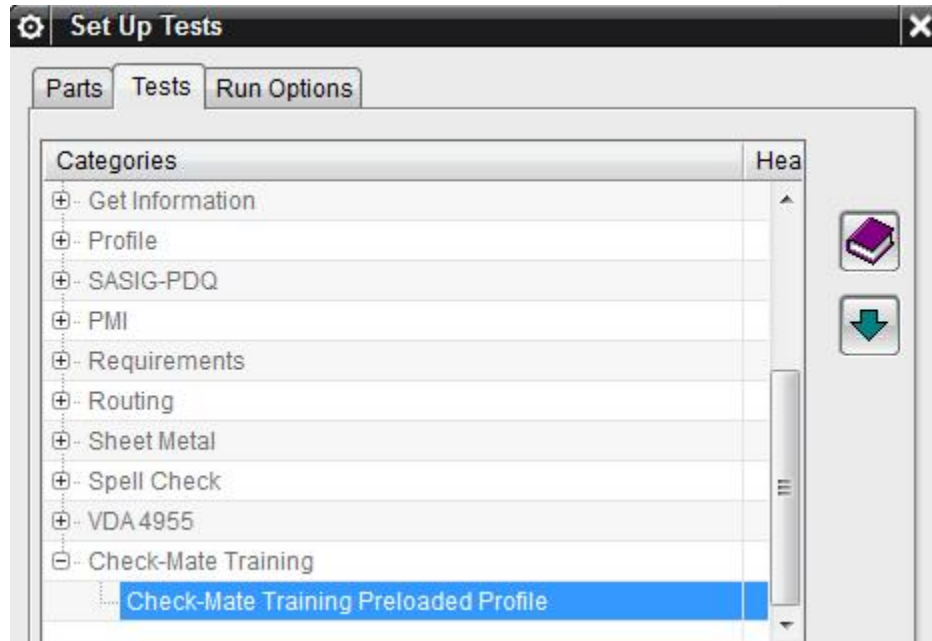
- ☐ Return to the “**Tests**” tab in the Check-Mate user dialog.
- ☐ Select all of the tests in the lower **Chosen Tests** window. (Select the top check and then hold down the Shift button and select the bottom check.)
- ☐ Choose the **Remove from List** button on the right edge of the dialog.



- ☐ The tests will be removed from this lower window, leaving the window completely empty for now. (We'll add something back in the next step.)

Step 2: Find the Desired Profile and examine the contents

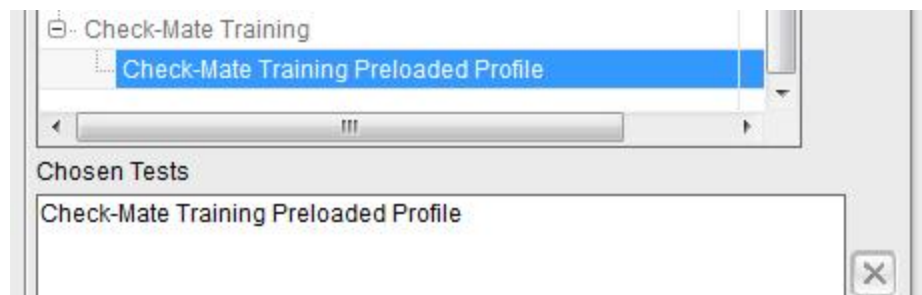
- ☐ Expand the **Check-Mate Training** node in the list of available checks.
- ☐ Choose the **Check-Mate Training Preloaded Profile** from the list and examine the contents by choosing the Documentation button on the right edge of the dialog.



- ☐ The information window appears and describes the checks present in this profile. The contents of this information window are customizable, as you will see later.

Step 3: Add the desired Profile to the list of Chosen Tests

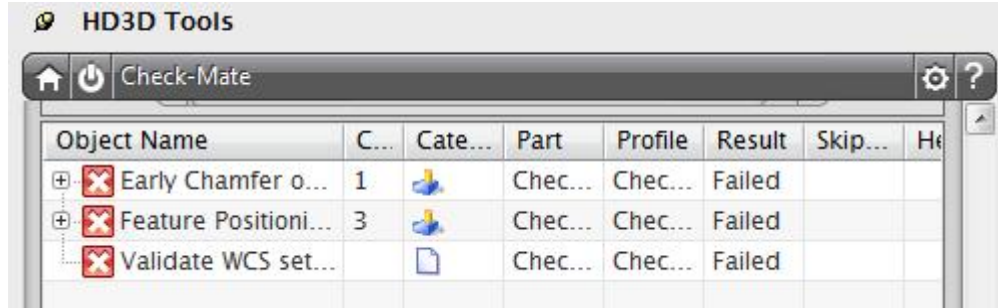
- ☐ Drop this profile into the **Chosen Tests** window using the green **Add to Selected** arrow.



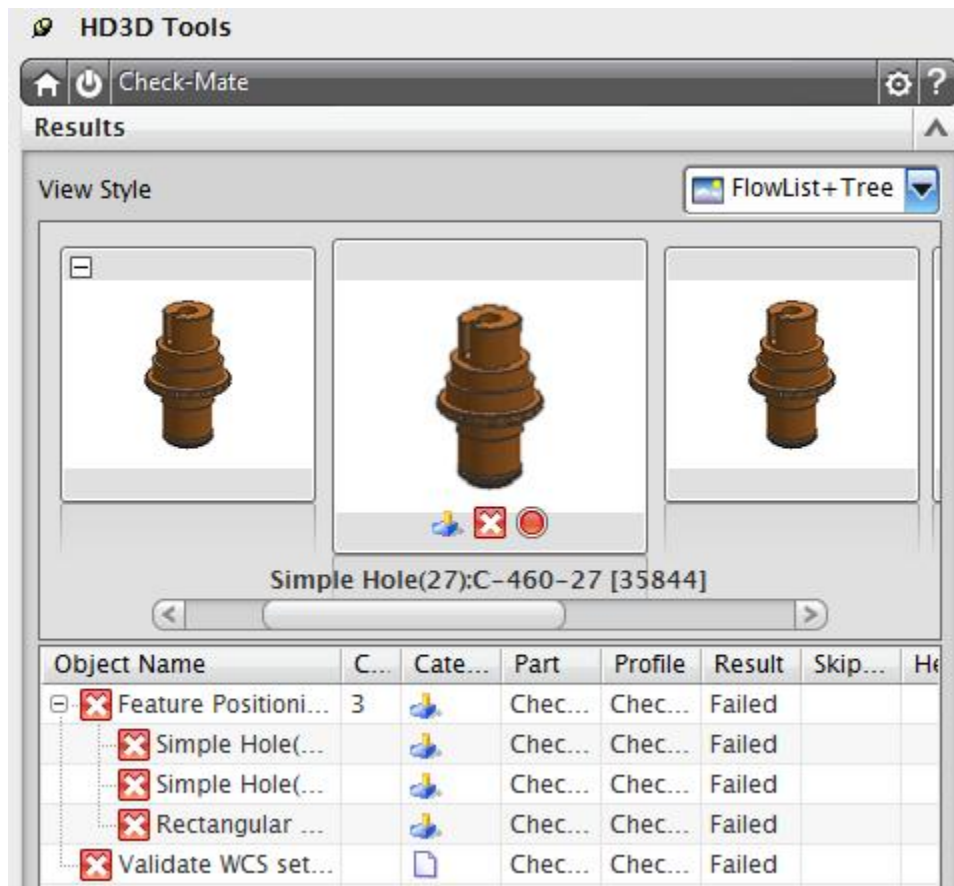
Step 4: Run the selected profile against the selected part(s).

- ☐ Once your list is complete, proceed to the third tab, **Run Options**.
- ☐ Leave all selections at their default state for now. Near the bottom, press the **Execute Check-Mate** button to run the selected checks against the selected parts.

Step 5: Examine the results.



- ❑ Expand the tree as shown to find the two remaining errors.



The checks were selected individually, but now they are grouped under the title **Check-Mate Training Preloaded Profile**.

Creating a profile consolidates a set of desired checks into one item in the user dialog, simplifying the interaction needed to find, configure, and execute them.

Step 6: Identify and correct the second and third failed checks.

**Step 7: When you have completed this activity, close your NX session.
(File → Exit) There is no need to save this test part.**

[End of Activity]

Deploying Custom Checks into an NX Session

To review, where did this custom **Check-Mate Training Preloaded Profile** come from? How did NX know to load it into this NX session?

Again, the main environment variable that helps NX know where to look for Check-Mate classes is **UGCHECKMATE_USER_DIR**. NX will recursively search for any checks it can find in the folder referred to by this environment variable.

You may remember that we set this environment variable in the batch script you used to start NX or using NXCustom. NX searched in the **UGCHECKMATE_USER_DIR** folder, found a DFA file (Knowledge Fusion source code file) and loaded the profile into your Check-Mate session as soon as you entered the Check-Mate dialog.

The next few activities will illustrate the function of other important environment variables that can be added to the NX startup (or used as system variables, etc.) with NX Custom.

Activity 4 – Configuring Checker Visibility

In this activity we will remove all OOTB checkers and profiles from the user interface, leaving only the **Check-Mate Training Preloaded Profile**. This will demonstrate one tool for enforcing the execution of authorized checks with standard settings.

Step 1: Set the desired category to be the only one visible.

☐ Open Windows Explorer and navigate to the location of NX Custom and the NX*env.dat file.

☐ At the bottom of the file uncomment the line that reads:

```
# UGCHECKMATE_ALLOW_CATEGORY=Check-Mate Training
```

...by removing the # statement at the beginning of the line. The line should then read:

```
UGCHECKMATE_ALLOW_CATEGORY=Check-Mate Training
```

This will tell NX to display (allow) ONLY the category called **“Check-Mate Training”**

☐ Save this ENV file.

Step 2: Start NX and start a new, blank part.

- ☐ Start NX using NXCustom. The UGCHECKMATE_ALLOW_CATEGORY environment variable will be set, and NX started using this new configuration.
- ☐ When NX comes up, start a new, blank part.

Step 3: Test the visibility of categories in the Check-Mate dialog.

- ☐ Go into the Check-Mate dialog.
- ☐ Only the “**Check-Mate Training**” category and the **Check-Mate Training Preloaded Profile** contained within are visible.

NOTE: If other checkers were present in this category, they would also be visible in this mode. Remember that this visibility filter is being applied to categories, and not individual checkers. If you would like to change the visibility of an individual checker, you can use the UGCHECKMATE_ALLOW_CHECKER and UGCHECKMATE_HIDE_CHECKER environment variables.

Step 4: When you have completed this activity, close your NX session.

[End of Activity]

Activity 5 – Pre-populating the Chosen Tests Area

In this activity we will pre-populate the Chosen Tests area in the Check-Mate user dialog to further accelerate and remove ambiguity from the user experience.

Step 1: Set the desired profile to be the default checker.

- ☐ Open Windows Explorer and navigate to the location of NXCustom and the NX*env.dat file.
- ☐ Uncomment the line containing the UGII_CHECKMATE_DEFAULT_CHECKER environment variable so that it reads (all on one line):

```
UGII_CHECKMATE_DEFAULT_CHECKER=  
cm_train_preloaded_profile
```

NOTE: This environment variable is expecting to use the **Class Name** for the profile, as opposed to the **Display Name**. The class name will not contain spaces and may be a slightly more cryptic text string. They will often be all lower-case letters, as well.

- ☐ Save this ENV file.

Step 2: Start NX and start a new, blank part.

- ☐ Start NX using NXCustom. The UGCHECKMATE_DEFAULT_CHECKER environment variable will be set, and NX started using this new configuration.
- ☐ When NX comes up, start a new, blank part.

Step 3: Test the pre-population in the Check-Mate dialog.

- ☐ Go into the Check-Mate dialog.
- ☐ The **Check-Mate Training Preloaded Profile** should already be visible down in the “Chosen Tests” area of the dialog.

NOTE: If you would like to display more than one checker or profile in the Chosen Tests field, then class names can be separated by a comma when this environment variable is declared.

- ☐ Do not close your NX session just yet...

[End of Activity]

Activity 6 – Invoking the Profile Automatically

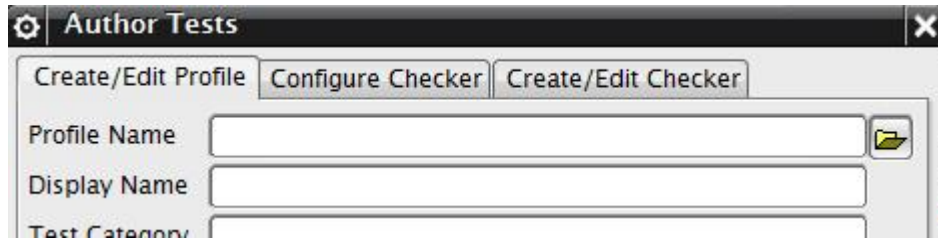
In this activity we will modify the profile to allow it to be automatically run at save time. We will also “turn on” the ability to automatically run Check-Mate in an NX session. This will further streamline the user interaction and enforce execution of this profile when parts are saved.

Step 1: Open the Check-Mate Author dialog.

- ☐ Use the NX command finder or go to Analysis → Check-Mate → Author Tests in the pull-down menus.
- ☐ The first tab in this dialog is for creating or editing profiles – and that’s what we’re going to do now.

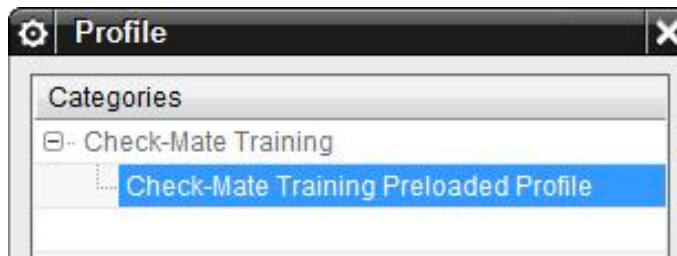
Step 2: Edit the profile to run at Save-Time.

- ☐ Call up your **Check-Mate Training Preloaded Profile** by choosing the **Browse...** (folder) button at the right end of the top line.



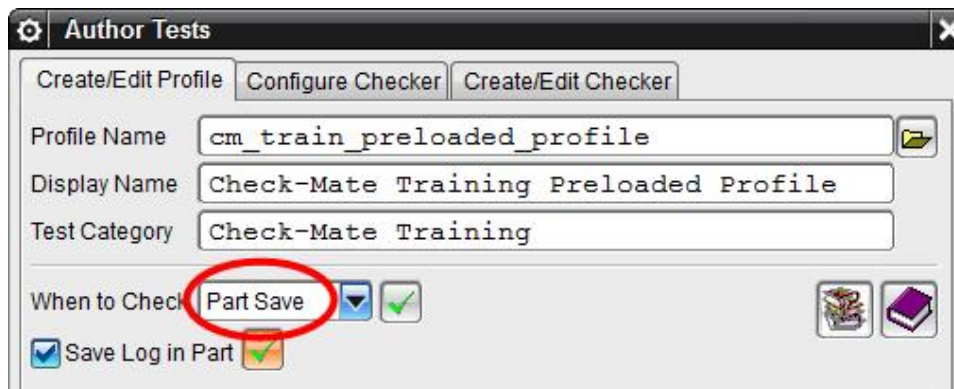
The little window that comes up allows you to browse through all of the loaded profiles and open them here for editing.

- ❑ Select your **Check-Mate Training Preloaded Profile** in the **Check-Mate Training** branch of the tree and choose **OK** to load this profile into the dialog. If you continued from the previous activity, this should be very simple, as this may be the only profile currently loaded and visible.



NOTE: The **Chosen Profiles and Checkers** area should now contain the five checks contained in this profile. You can use this technique to re-open up and re-save your profiles as you work on them.

- ❑ From the **When to Check** option menu, choose the “**Part Save**” option.



Step 3: Save the modified profile.

- ❑ Choose the **Save Profile As** button at the bottom of the dialog.

- ☐ Navigate to the NXCustom directory and \Checkmate and save the profile there as “cm_train_preloaded_profile.dfa”. This name should actually be pre-populated in the **Save DFA File** dialog for you. Also the **Checkmate** directory will also probably be chosen as the default directory.

NOTE: As a best practice, Knowledge Fusion classes (like this profile class we’re saving here) should be saved using the same name for the dfa file as the class contained inside the file. In this case, the class name for your profile was entered up in the top line of this dialog. Notice that NX has automatically suggested an appropriate name for the profile dfa file.

Step 4: When you have saved the profile, close your NX session.

Step 5: Confirm in the DFA file that the change has been made.

- ☐ Navigate to the NXCustom directory and \Checkmate and open the file there called “cm_train_preloaded_profile.dfa”. This file contains the actual Knowledge Fusion code that defines this profile.
- ☐ About halfway down the file there is a line that should read:

```
( Integer )      check_time_index: 3;
```

This check_time_index: attribute controls the timing of the execution of this profile. A value of 1 corresponds to manual execution. A value of 3 (as shown here) corresponds to automatic checking at save time.

- ☐ In the off chance that this line does not read as above, change it now so that it does, and save this file.

NOTE: When deploying Check-Mate, it is extremely valuable to have some understanding of Knowledge Fusion syntax. As you can see, the Check-Mate dialogs are really just a GUI wrapped around editing some underlying Knowledge Fusion code.

Step 6: Enable automatic running of Check-Mate.

- ☐ Open Windows Explorer and navigate to the location of NXCustom and the NX*env.dat file.

- ☐ Uncomment the line that reads:

```
# UGII_CHECKMATE_ALLOW_AUTO_RUN=1
```

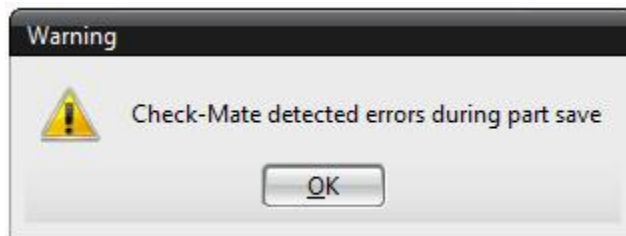
This will tell NX to enable automatic execution of your Check-Mate profile at the time specified in the profile.

- ☐ Save this ENV file.

Step 7: Start NX and open the part Check-Mate_part_01.prt.

Step 8: Test the auto-run check by saving the part.

- ☐ Choose the Save icon on the toolbar (or File → Save) to save this part.
- ☐ Notice the small error window that pops up, warning you that there are errors in this part.



- ☐ To review the errors, go to the results tab in the Check-Mate dialog.

Step 9: When you have completed this activity, close your NX session.

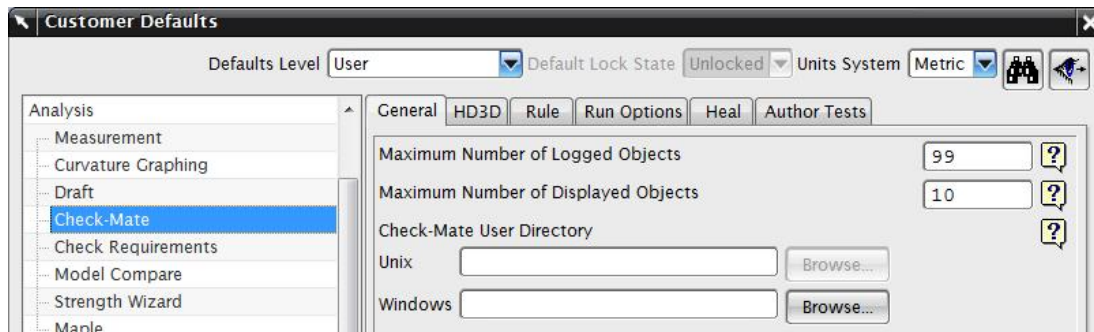
[End of Activity]

Section 2: Customer Defaults

General

The first two options in the group control the volume of results that will be reported when a check produces a large number of failing entities. (Some detailed topology checks like Tiny Edge will sometimes report a very large number of entities.)

The “Maximum Number of Logged Objects” default controls the number of entities that will be written into Teamcenter and/or external XML log files. It will not affect the display in the interactive NX session, though a large value here will increase the time NX takes to generate log files and/or write results to the Teamcenter database.



The “Maximum Number of Displayed Objects” default controls the number of entities that will be reported through the HD3D interface. (Thus, this also controls the maximum number of tags that will be generated by any one checker as well.) Conceptually, keeping this value at a reasonably small number (as shown) will generate a set of failures that the user can start fixing, and once these are fixed, the check can be re-executed to collect another set of entities to be repaired. In all cases, the total number of failures for each checker will be reported in the “Count” column, even if this number exceeds the “Maximum Number of Display Objects” customer default value, as shown here:

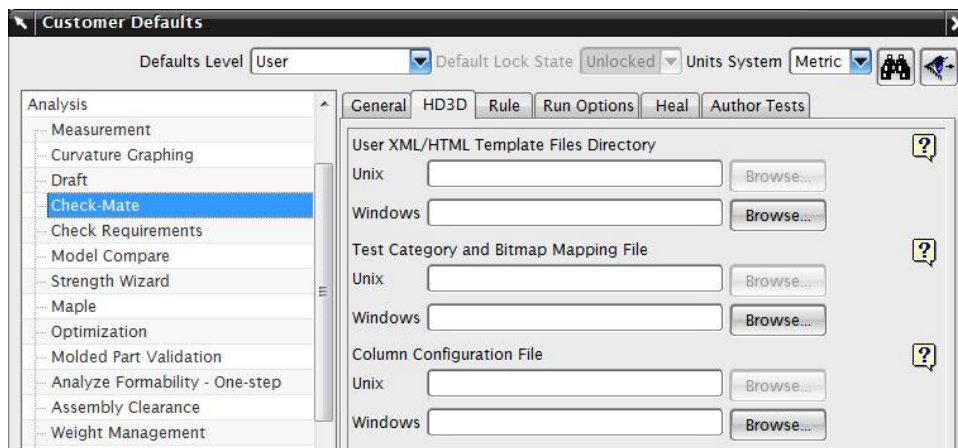
Object Name	Count	Cat...	Part
Tiny edge: G-ED-TI	14		b_side_main_c
Edge:R-8998-13 [36820]			b_side_main_c
Edge:R-9778-13 [35159]			b_side_main_c
Edge:R-8415-13 [37427]			b_side_main_c
Edge:R-7559-13 [36488]			b_side_main_c
Edge:R-9163-13 [37254]			b_side_main_c
Edge:R-9162-13 [35491]			b_side_main_c
Edge:R-10208-13 [35730]			b_side_main_c
Edge:R-7666-13 [37153]			b_side_main_c
Edge:R-7575-13 [35411]			b_side_main_c
Edge:R-7389-13 [37632]			b_side_main_c

(Ten objects are displayed, and the “Count” column reflects fourteen total failures.)

HD3D

Templates Location

The presentation of Check-Mate results, including the contents of tooltips and the HD3D Info tool can be configured. The format of these presentation elements is controlled by a set of XML and HTML templates. By default, these templates are located in the \DESIGN_TOOLS\checkmate\customization folder in the NX install. If you decide to edit these templates, then use the customer defaults on the HD3D tab to tell NX where to find your edited template files.



Checker Category Bitmaps

With HD3D, a checker category icon can be seen in the tree list view style, the flow list view style, and the tile list view style. These bitmaps have been pre-configured for all out-of-the-box checkers using the *checker_bitmap.xml* file in the \DESIGN_TOOLS\checkmate\customization folder. If you would like to display icons for your custom checkers and/or profiles or if you would like to change the icons shown for out-of-the-box checkers, this can be accomplished by editing a copy of the *checker_bitmap.xml* file and then using the second customer default on the HD3D tab to locate your new mapping file.

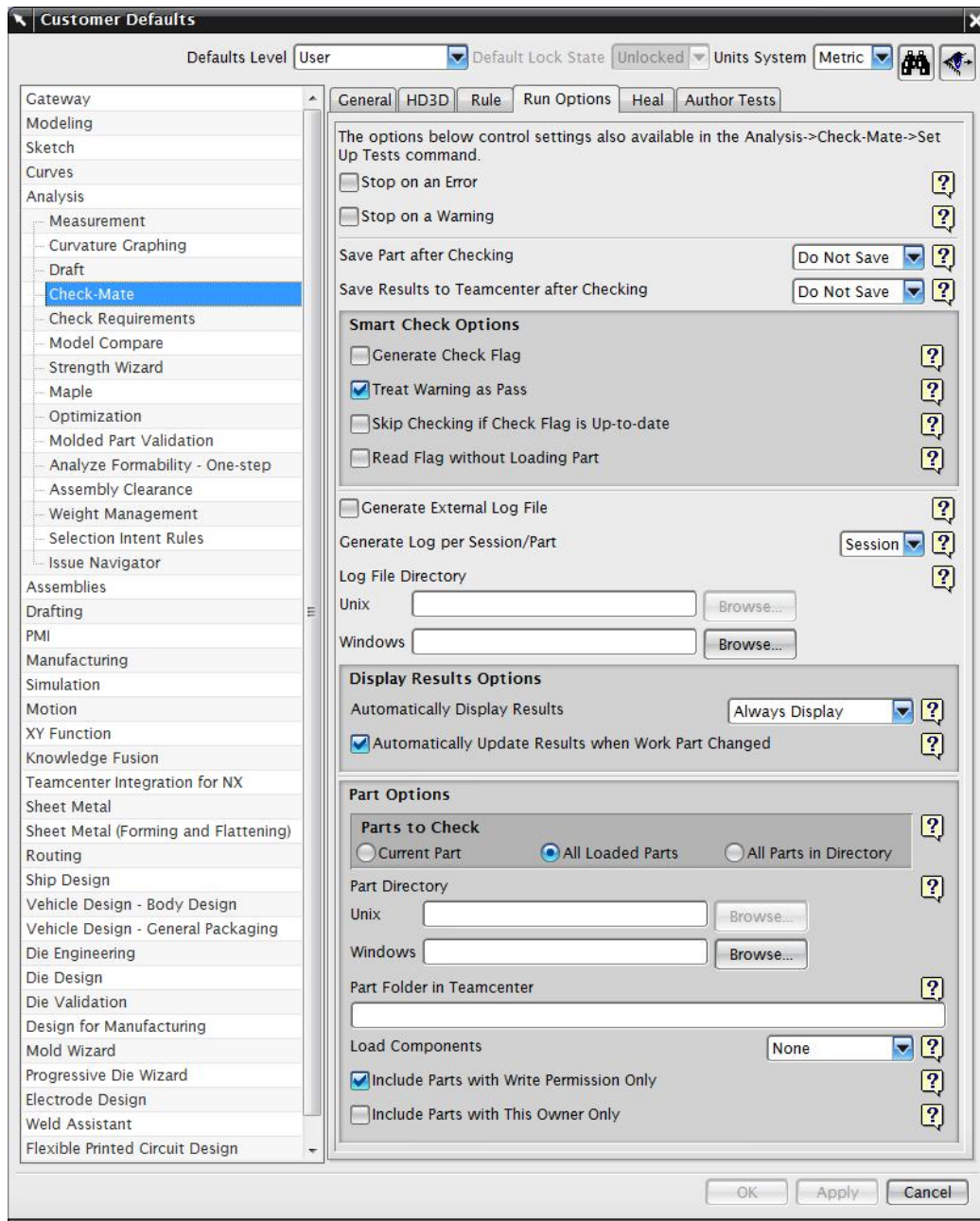
HD3D Columns

Similarly, the columns in the HD3D Tree presentation can also be configured using the third customer default here and a copy of the *column_configuration.xml* file normally found in the same \DESIGN_TOOLS\checkmate\customization folder.

Run Options

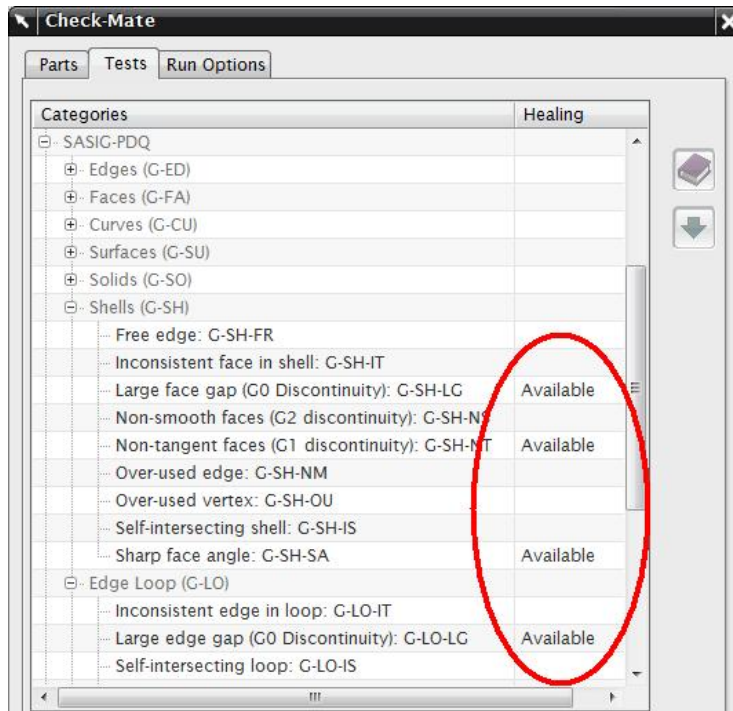
All of the options in Check-Mate *Set Up Tests* dialog can be pre-configured using the Customer Defaults. In this way, desired Check-Mate behavior can be pre-configured and deployed across a group of users.

In particular, it is important to consider the desired behavior around whether or not (and where) to save external log files, whether or not to save parts immediately after checking, and which parts to check by default (all loaded parts versus only the current work part, parts to which the user has read-only access, or only parts that the user can actually affect, etc.) This behavior will vary from company to company, but is key to making the Check-Mate experience as seamless and simple as possible for end users.



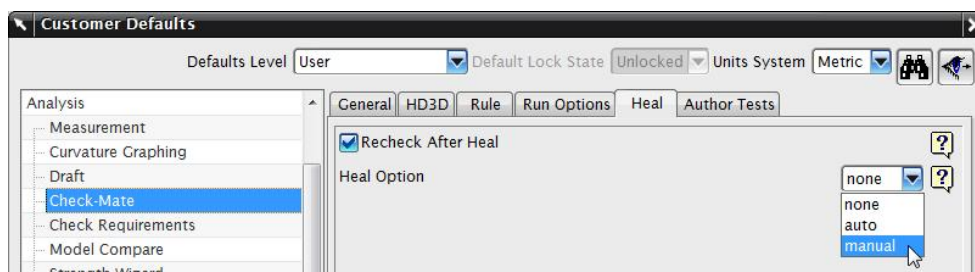
Heal

Certain checkers contain code that can actually heal the problems discovered by the checker. Most of the checkers that contain healing code out-of-the-box are part of the SASIG-PDQ geometry and topology fidelity checking category, and these checkers are indicated by the word “Available” in the “Healing” column in the “Tests” tab of the Check-Mate *Set Up Tests* dialog:



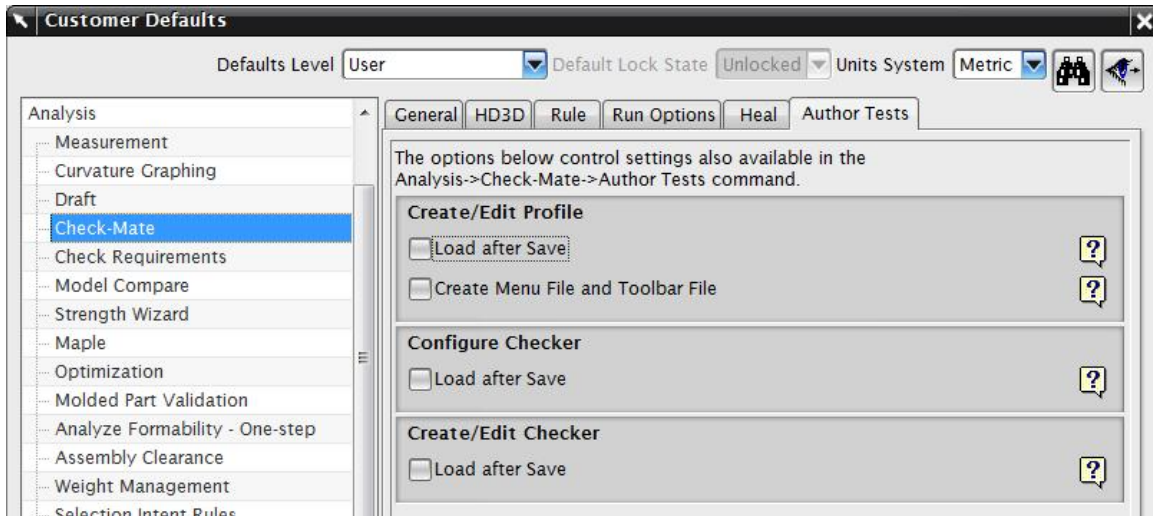
The customer defaults for Check-Mate healing indicate whether Check-Mate should double-check the part after healing to make sure that the healing operation was in fact effective, and whether checking will be:

1. Not available at all (on the MB3 menu for HD3D tags and list objects in the dialog)
2. Available on the MB3 menu and attempted automatically after checking
3. Available for manual execution on the MB3 menu

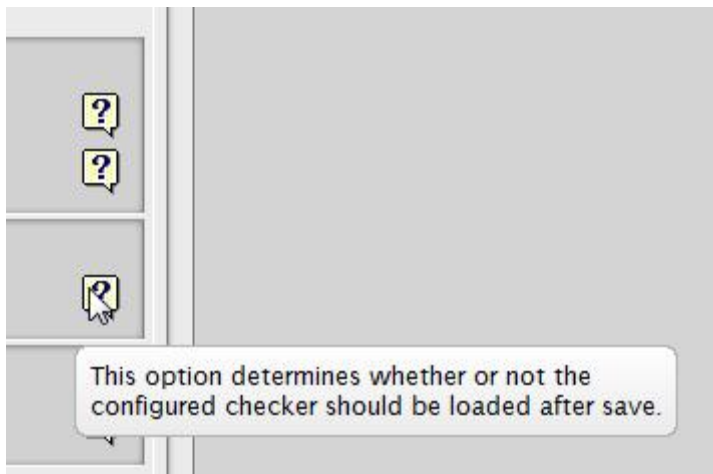


Author Tests

The settings on the “Author Tests” tab control the default settings for various options in the Author Tests dialog. This page should be fairly self-explanatory. ☺



Remember that mousing over each question mark icon on the left edge of the dialog will produce a tooltip describing that particular setting:



Section 3: Advanced Check-Mate Customization

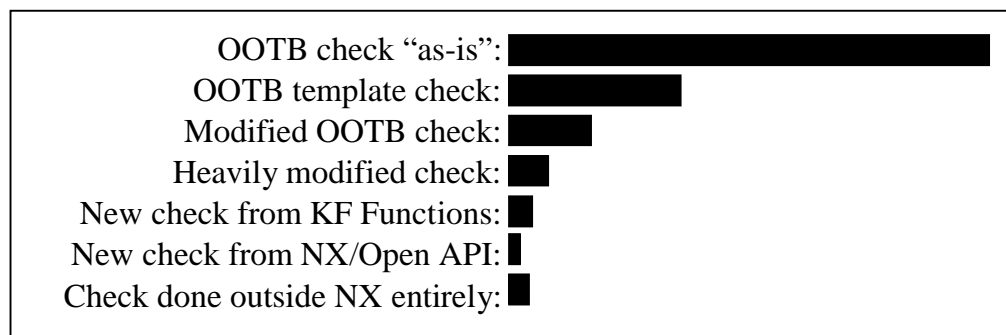
The Check-Mate Configuration Continuum

As engineering managers or system administrators start to decide how they will use Check-Mate within their organization, a common starting point is the set of existing company design standards.

Commonly, a gap analysis is done comparing this set of desired checks against Check-Mate capabilities and deciding which checks can be accomplished by:

1. using an out-of-the-box (OOTB) Check-Mate check “as-is”
2. providing input parameters to an OOTB check (template)
3. modifying an OOTB check in minor ways
4. significantly changing an OOTB check
5. creating an entirely new check using existing Check-Mate functions
6. creating an entirely new check using NX/Open API functionality
7. performing the check outside NX entirely

The following graph illustrates (in entirely relative terms) the frequency with which we see customers using these seven options.



While the OOTB checks (complete and template checks) seems to meet the vast majority of our customers’ general needs, there is a tremendous variety in the specific needs of our customers. In the end, however, the great breadth and depth of coverage afforded by Knowledge Fusion and the NX/Open API are able to almost always allow customers to perform the checking they want to do. As a result, modification of existing similar checks is fairly frequent.

After looking briefly at how users will interact with Check-Mate, we will look at how you (as a site administrator) can pre-configure the Check-Mate environment to be even more productive for users at your company.

Activity 7 – Creating a Check-Mate Profile

The first step along the Check-Mate continuum mentioned in the first section of this class is identifying OOTB checks that already address specific checking needs “as-is”. The second step is identifying “template” checks that are very close, but may need a bit of additional input to function correctly.

In this activity, you will learn how to search for, find, configure, and incorporate OOTB checks into a new custom profile. We will save this profile into our pre-defined custom location described above.

For the purposes of this lesson, we will assume that your current model checklist includes the following items:

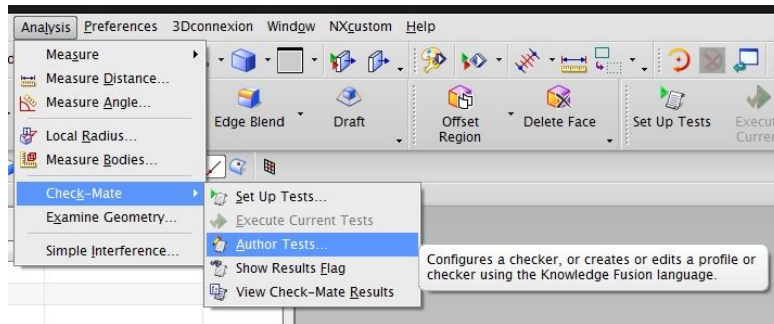
Check #	Description	Desired Status
1	Sketches should be fully constrained	FAIL
2	Sewn solids should require a sewing tolerance no larger than 0.01mm	WARNING
3	Parts should not contain unparameterized features	WARNING
4	Part must contain two reference sets called MODEL and FACET	FAIL
5	Parts should not contain suppressed features	FAIL

Step 1: Check the NX Env variables. Open a new, blank part.

- ☐ We need to display all the Checks again, so you may need to comment out the UGCHECKMATE_ALLOW_CATEGORY variable using NXCustom and restart NX.
- ☐ We then need a part open to get to the Check-Mate dialogs. A blank part is sufficient, though.

Step 2: Open the Check-Mate Author dialog.

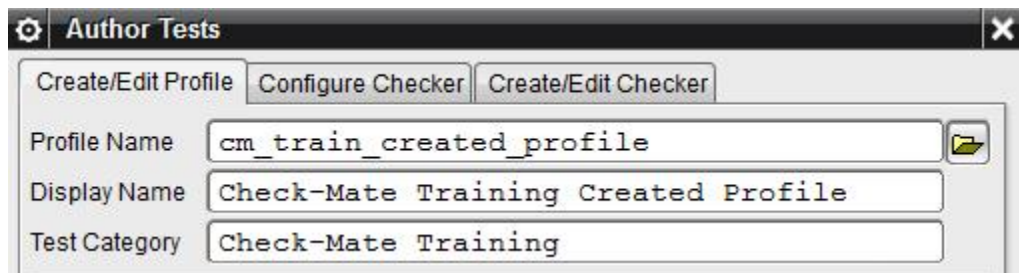
- ☐ Use the second button (**Author Tests**) on the Check-Mate toolbar, or go to Analysis → Check-Mate → Author Tests in the pull-down menus.



- ☐ The first tab in this dialog is for authoring profiles – and that’s what we’re going to do now.

Step 3: Enter the required information to create a new profile.

- ☐ Enter “**cm_train_created_profile**” in the Profile Name line, as seen below. This name is a Knowledge Fusion **class name**, and must not contain spaces. Underscore characters and dashes are fine as separators.
- ☐ Enter “**Check-Mate Training Created Profile**” (without underscores) in the Display Name line, as seen below. This is the name that users will see in the Check-Mate Run tests dialog, so this one should be as descriptive as possible. Spaces are okay in this field.
- ☐ Enter “**Check-Mate Training**” (without underscores) in the Test category line, as seen below. This is the name of the category under which your new profile will appear.



NOTE: If you use the name of an existing category, your profile will appear under that category. If you use a new name here, a new category will be created automatically for you. You can descend (or create) category hierarchy using periods. For example, if you wanted to create a subcategory under “Check-Mate Training” called “profiles” you could enter “Check-Mate Training.profiles” here, and that structure would be created automatically for you.

Step 4: Generate the list of available checkers.

- ☐ Right-click in the area where the word **Categories** appears and choose “Display checkers in Information Window”.



- ☐ The listing window will display the list of all checkers currently contained in this tree structure.

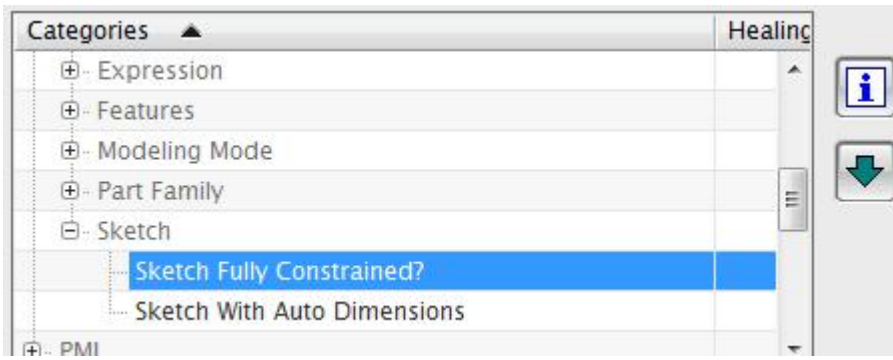
NOTE: The option to “Display profiles in Information Window” will display a similar list of all displayed profiles.

Step 5: Search for checks related to “sketch”.

- ☐ In the listing window, choose **Edit → Find** from the pull-down menus or press **ctrl+F** to open the **Find** window.
- ☐ Enter the word “sketch” here, and hit Enter or click on the **Find Next** button.
- ☐ Find a checker that determines whether sketches are fully constrained. (Sketch Fully Constrained?, or %mqc_check_sketch_fully_constrained)
- ☐ Note the location of this profile, based on the leftmost column in the listing window.

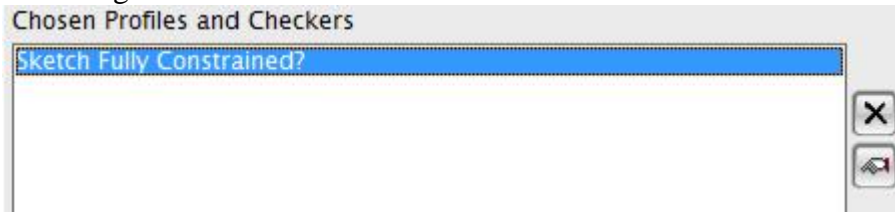
Step 6: Add the sketch constraints check to the profile.

- ☐ Expand the tree in the Check-Mate Author Checks dialog to find the sketch constraints check.
- ☐ Add the sketch constraints check to the **Chosen Profiles and Checkers** window below by selecting the check and choosing the **Add to List** green arrow button on the right.



☐ Congratulations. You've added your first check to a custom profile.

- ❑ To set the severity of the result to FAIL, select the checker in the LOWER list and choose the **Customize Test** on the right edge of the dialog.



- ❑ Inside the Customization dialog, confirm that the **Log Option** has been set to **LOG_ERROR**. This indicates the most severe result that can be obtained from Check-Mate. We'll use the WARNING status in just a minute as well.



NOTE: This dialog lets you set preferences and default values for individual checks. As we save these checks in a profile, the settings you specify here will also be saved with the profile. In this way, you can pre-configure the checks in your profile and ensure that everyone in the company is checking their parts against a consistent standard.

Step 7: Search for checks related to “sew”.

- ❑ In the listing window showing the list of currently available checks, choose **Edit → Find** from the pull-down menus or press **ctrl+F** to open the **Find** window.

- ☐ Enter the word “sew” here, and hit Enter or click on the **Find Next** button.
- ☐ Find a checker that determines whether sewn bodies can be successfully sewn using the specified sew tolerance. (Check Sewing Tolerance, or %mqc_check_sewnsheet_tolerance)
- ☐ Note the location of this profile, based on the leftmost column in the listing window.

Step 8: Add the sew tolerance check to the profile.

- ☐ Expand the tree in the Check-Mate Author Checks dialog to find the sew tolerance check.
- ☐ Add the sew tolerance check to the **Selected Checkers and Profiles** window below by selecting the check and choosing the **Add to List** green arrow button on the right.
- ☐ To set the severity of the result to WARNING, select the checker in the **LOWER** list and choose the **Customize Test** on the right edge of the dialog.
- ☐ Inside the Customization dialog, confirm that the **Log Option** has been set to **LOG_WARNING**. This indicates a non-critical status for this error – it’s something that the designer (and/or manager) might want to be aware of, but it should not prevent the part from being released, for example.
- ☐ Also inside the Customization dialog, set the **Sew Tolerance** value to 0.01mm. Choose OK when you are done.

Step 9: Search for dumb geometry check.

- ☐ In the listing window showing the list of currently available checks, choose **Edit → Find** from the pull-down menus or press **ctrl+F** to open the **Find** window.
- ☐ What might you search for here to find a check that looks for “dumb” geometry within a part file? Try and find this check.

HINT: How does “dumb” geometry appear in the timestamped model history? What kind of feature represents this “dumb” solid or surface?

- ☐ Once you have found the checker (you’re looking for "Check Existence of Unparameteriz", or %mqc_check_unparam_feature, by

the way), note the location of this profile, based on the leftmost column in the listing window.

Step 10: Add new check to the profile.

- ☐ Expand the tree in the Check-Mate Author Checks dialog to find **Check Existence of Unparameterized Features**. (Inside Modeling → Features)
- ☐ Add this check to the **Selected Checkers and Profiles** window below by selecting the check and choosing the **Add to List** green arrow button on the right.
- ☐ To set the severity of the result to WARNING, select the checker in the LOWER list and choose the **Customize Test** on the right edge of the dialog.
- ☐ Again, inside the Customization dialog, confirm that the **Log Option** has been set to **LOG_WARNING**.

Step 11: Save the profile to the custom location.

- ☐ Let's save our work on this profile so far. First, make sure that the **Load After Save** option in the bottom left corner of the dialog is checked. This will ensure that your profile is immediately loaded into your NX session.
- ☐ Choose the **Save Profile As** button at the bottom of the dialog.
- ☐ Navigate to NXCustom\Checkmate and save the profile there as "**cm_train_created_profile.dfa**". This name and directory should be pre-populated in the **Save DFA File** dialog for you.

NOTE: As a best practice, Knowledge Fusion classes (like this profile class we're saving here) should be saved using the same name for the dfa file as the class contained inside the file. In this case, the class name for your profile was entered up in the top line of this dialog. Notice that NX has automatically suggested an appropriate name for the profile dfa file.

[End of Activity]

Activity 8 – Using a Template Check

The next check we are going to add will be a template check. This is a check delivered OOTB that requires more input before it can really do anything. A few template checks can be used like the sew tolerance check in the last activity (by just specifying some input in a customization dialog) but many will require a copy of the template check to be saved to disk. We'll use one such check here so that you can see how this works.

In this activity, we will go straight to the environment where you can edit and save copies of OOTB checks. We'll use this same environment again in the next activity to get started when we make major changes to a check.

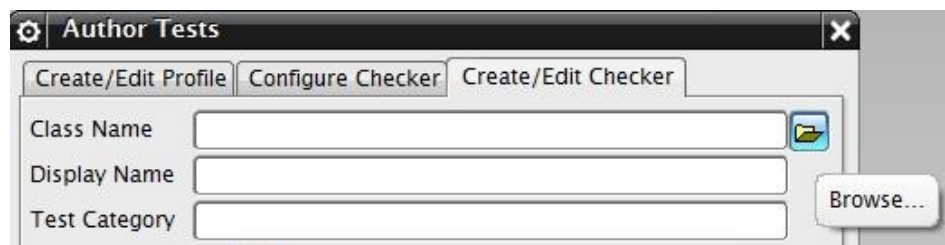
Check #4 in the sample list of requirements is looking for a couple of required reference sets. If you were to search for this check in the same way as the others, you'd find a check called **Check Multiple Required Reference Sets** in the **Template.File Structure** branch of the tree. Because we've found this checker in the **Template** branch, we're going to approach it in a slightly different way. We're going to save a new copy of this check that contains our required specific inputs.

Step 1: Go to the Create/Edit Checker Tab in the Check-Mate Author Dialog

- ☐ This third tab in the Author Tools dialog lets you edit existing checkers. You'll use this tab now to specify input for this template check.

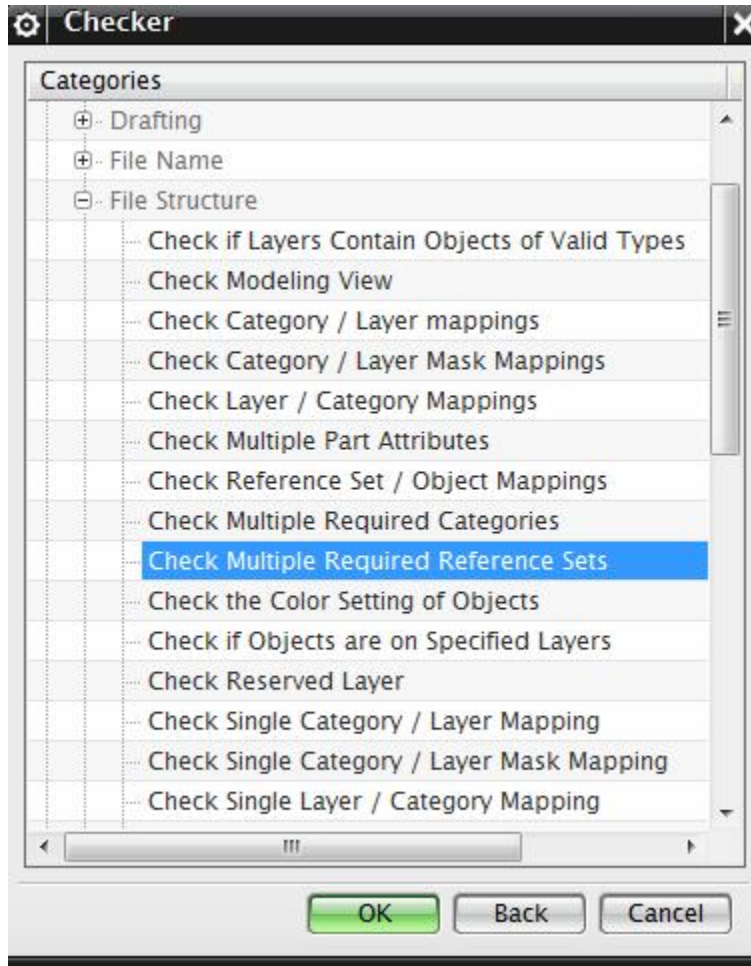
Step 2: Browse for the desired template check.

- ☐ Choose the **Browse...** button on the right edge of the dialog.



The little window that comes up allows you to browse through all of the loaded checks (template checks and all other checks) and open them here for editing. We will choose the check we want, change a few things, and then save a new copy into your custom location.

- ☐ Browse to the **Template.File Structure** branch of the tree and choose the **Check Multiple Required Reference Sets** checker.



- ☐ Choose **OK** to bring the contents of this check back to the dialog.

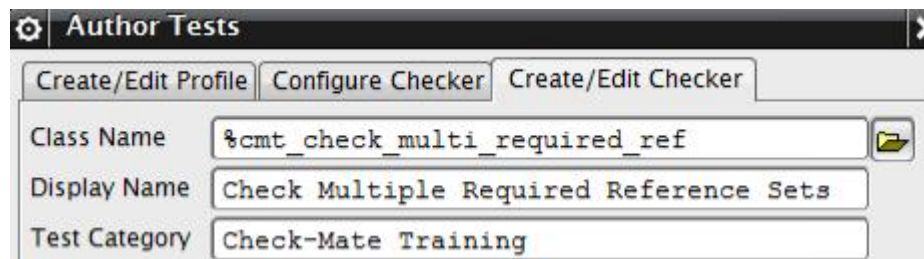
Step 3: Change the name of the template checker.

Now we want to create a custom version of this template check, and the most important step in doing so is to give the check a unique name. The name that must be unique (within your NX session) is the Class Name in the first line of the dialog, so...

- ☐ Edit the **Class Name** by changing the first few letters from the default “mqc” (Model Quality Check, in case you were wondering) to “cmt”, as shown below. This will help the name be unique.

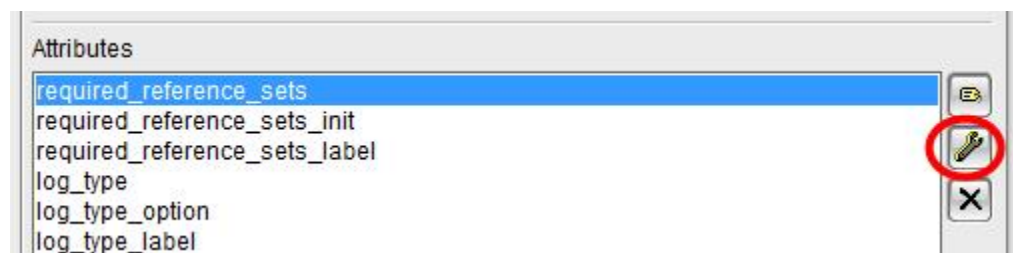
NOTE: At your company, you will likely want to standardize very early on a standard prefix that you will use for your company-specific checking classes. This will help prevent confusion in the future.

- ❑ You can leave the **Display Name** the same if you like. Display names can be duplicated in the tree, but to keep things understandable, we will put our new check in a different category.
- ❑ Edit the **Test Category** from the default “**Template.File Structure**” to be “**Check-Mate Training**”, as shown below. This will create a new folder called Check-Mate Training and place this new check inside it.

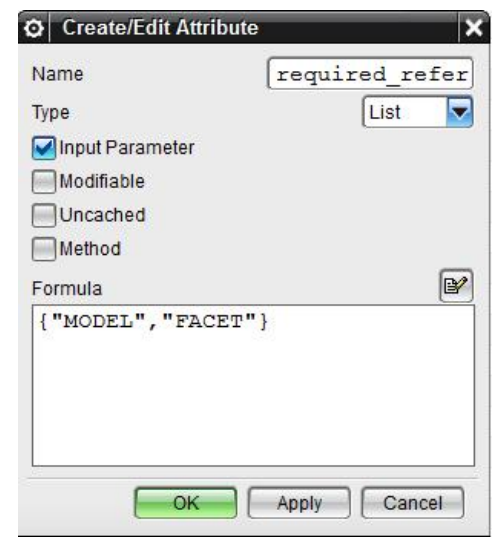


Step 4: Edit the appropriate input value.

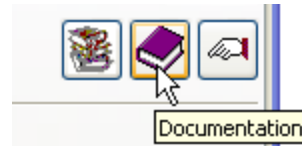
- ❑ Partway down the dialog is a list of input attributes for this checker class. Select the **required_reference_sets** attribute and choose the **Edit Attribute** button the right edge of the dialog.



- ❑ In the **Create/Edit Attribute** dialog that appears, add the two required reference sets, formatted as a list of strings, as shown.
- ❑ Repeat this task for the **required_reference_sets_init** attribute, adding the same list of reference sets there.



NOTE: It is always important to enter input attributes using the correct syntax. If you are ever wondering what format a particular attribute is expecting, the **Documentation** button just above the attribute list can help you find out.



Step 5: Save the modified check to disk.

- ☐ First, make sure that the **Load After Save** option in the bottom left corner of the dialog is checked. This will ensure that your new checker is immediately loaded into your NX session.
- ☐ Choose the **Save Checker As** button near the bottom of the dialog.
- ☐ Navigate to `NXCCustom\Checkmate` and save this check there as **“cmt_check_multi_required_ref.dfa”**. This name and directory should be pre-populated in the **Save DFA File** dialog for you.

NOTE: Again, as a best practice, Knowledge Fusion classes (like this profile class we’re saving here) should be saved using the same name for the dfa file as the class contained inside the file. In this case, the class name for your profile was entered up in the top line of this dialog. Notice that NX has automatically suggested an appropriate name for the profile dfa file.

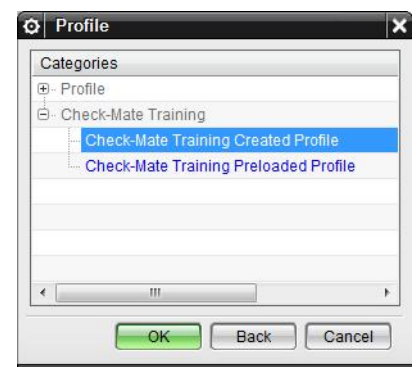
Step 6: Go back to the Create/Edit Profile Tab and load your custom profile.

- ☐ Call up your Check-Mate Training Profile by choosing the **Browse...** button at the right end of the top line.

The little window that comes up allows you to browse through all of the loaded profiles and open them here for editing.

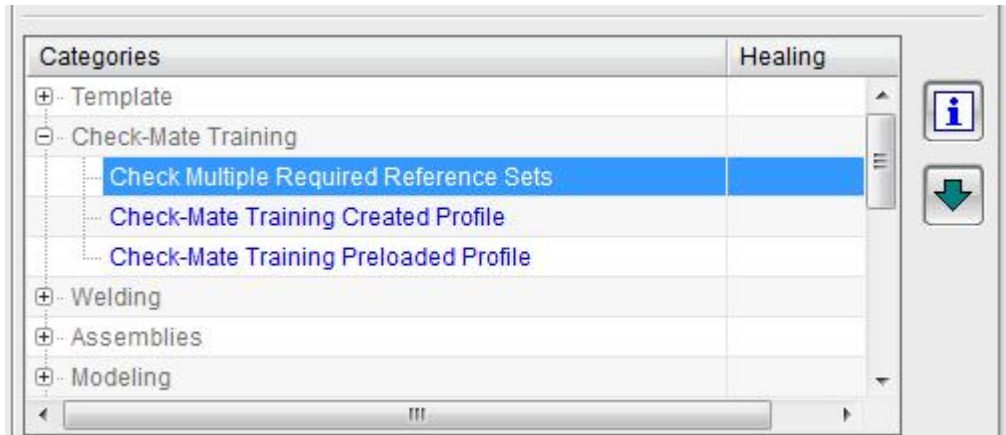
- ☐ Select your **Check-Mate Training Created Profile** in the **Check-Mate Training** branch of the tree and choose **OK** to load this profile into the dialog.

NOTE: The **Chosen Profiles and Checkers** area should now contain the three checks you had previously saved. You can use this technique to re-open up and re-save your profiles as you work on them.

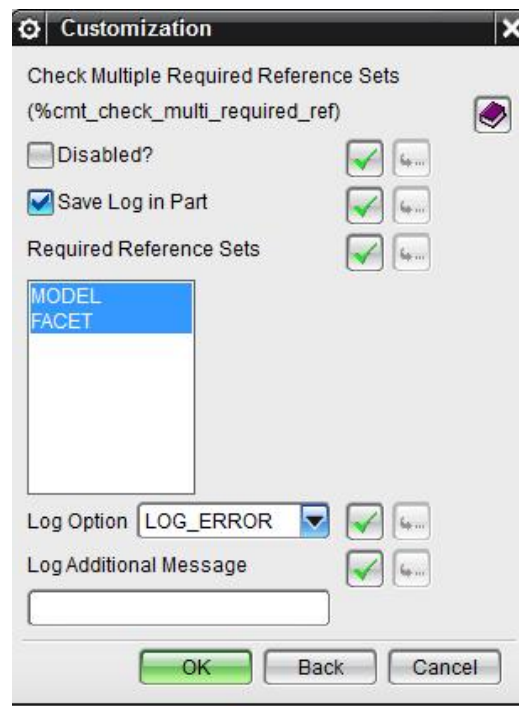


Step 7: Add the new check to the profile.

- ❑ Choose your new check out of the **Check-Mate Training** Checks category, and add it to your profile using the green arrow **Add to List** button on the right edge of the dialog.



- ❑ To confirm that your custom check contains the reference sets you specified, select your new check in the lower list and choose the **Customize Test** button on the right.
- ❑ Notice here that the **MODEL** and **FACET** reference set names have been added in the appropriate spot for this check.

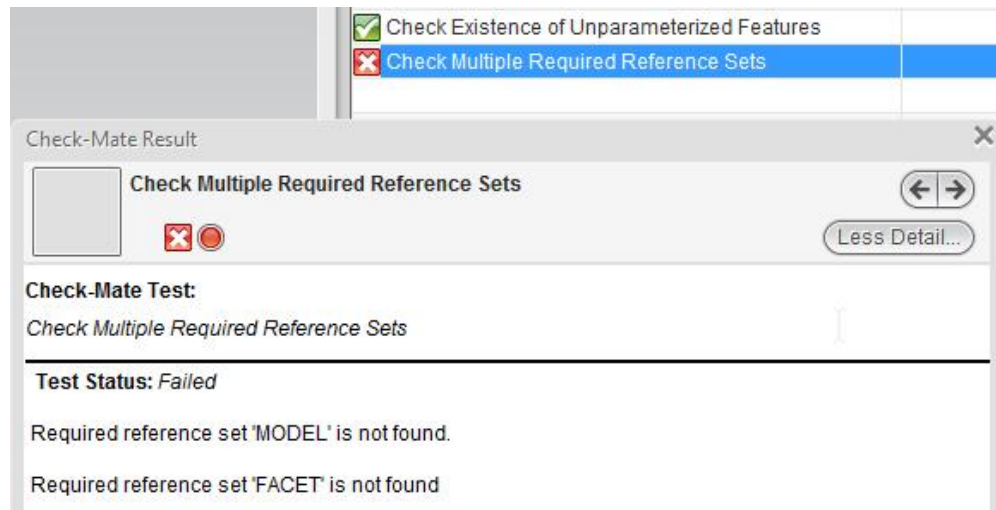


Step 8: Save the updated profile.

- ☐ First, make sure that the **Load After Save** option in the bottom left corner of the dialog is checked. This will ensure that your profile is immediately loaded into your NX session.
- ☐ Choose the **Save Profile As** button at the bottom of the dialog.
- ☐ Save as “**cm_training_created_profile.dfa**”. This name and directory should again be pre-populated in the Save DFA File dialog for you.

Step 9: Test the profile.

- ☐ Go to the Check-Mate Run Tests dialog and test your new profile. If you are using an unsaved empty part, this reference set check should fail, missing both reference sets.



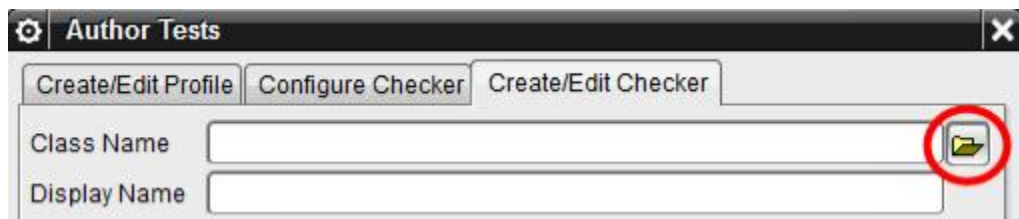
[End of Activity]

Activity 9 – Modifying an Existing Check to Create a New Check

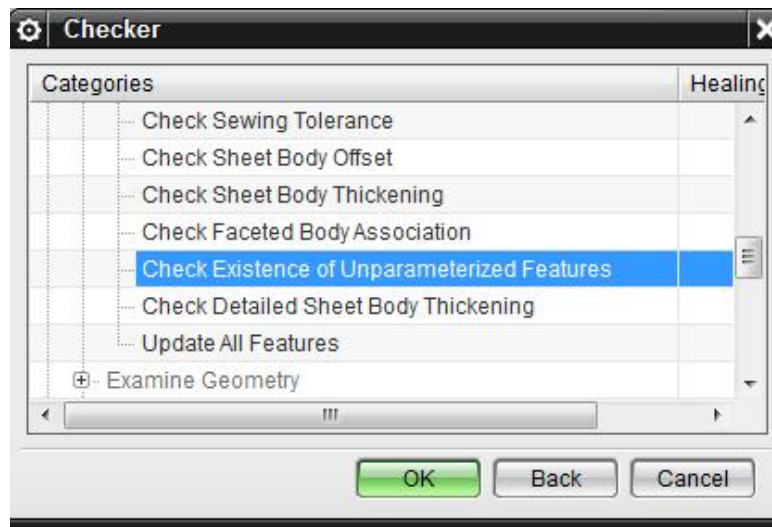
In this activity we will save the Knowledge Fusion code for an existing checker to disk and then edit the checker manually to create a new check. This will expose you more directly to the underlying Knowledge Fusion code used to create and customize checks. We will start with a checker that looks for Unparameterized Features and change it to look for Suppressed Features instead.

Step 1: Go to the Create/Edit Checker Tab in the Check-Mate Author Dialog

Step 2: Browse to find the desired existing check.



- ☐ Choose the **Browse...** button on the right edge of the dialog.
- ☐ Browse to the **Modeling.Features** branch of the tree and choose the **Check Existence of Unparameterized Features** checker.



- ☐ Choose **OK** to bring the contents of this check back to the dialog.

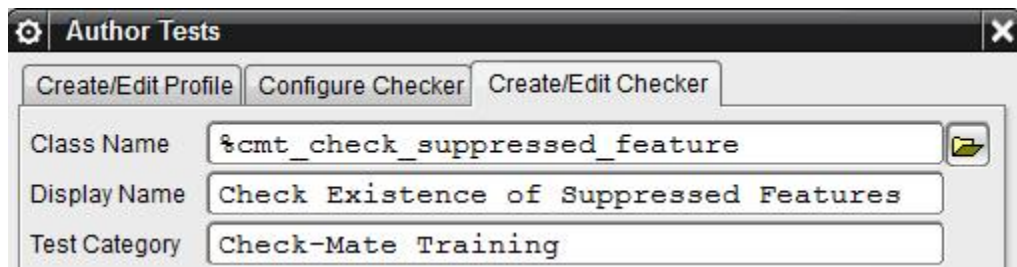
Step 3: Change the name of the template checker.

Remember that when creating custom checks it is very important to give each check a unique name. The name that must be unique (within your NX session) is the Class Name in the first line of the dialog, so...

- ☐ Enter a new **Class Name** by entering a new name, as shown below. This will help the name be unique. Remember that this class name should not have spaces in it. Check-Mate will warn you if you forget.

NOTE: At your company, you will likely want to standardize very early on a standard prefix that you will use for your company-specific checking classes. This will help prevent confusion in the future.

- ☐ Change the display name as well as shown below.
- ☐ Edit the **Test Category** from the default “**Modeling.Features**” to be “**Check-Mate Training**”, as shown below. This will assign this new check to the desired folder.



Step 4: Save the renamed check to disk.

- ☐ Choose the **Save Checker As** button near the bottom of the dialog.
- ☐ Navigate to `NXCUSTOM\checkmate` and save this check there as “**cmt_check_suppressed_feature.dfa**”. This name and directory should be pre-populated in the **Save DFA File** dialog for you.

Step 5: Edit the code as needed.

- ☐ Open the file “**cmt_check_suppressed_feature.dfa**” in your favorite text editor and examine it. Note that:

The name of the class (following `DefClass:`, in the third line) is the name that you entered in the Class Name field in the Check-Mate Author UI.

The `%test_category:` attribute has been populated with your entry from the “Test Category” field in the Check-Mate Author UI.

The `%displayed_name:` attribute has been populated with your entry from the “Display Name” field in the Check-Mate Author UI.

The `log_type:` attribute is currently set to 1. This corresponds to the FIRST entry in the next attribute, the `log_type_option:` list. The first entry is LOG_ERROR, so this check is currently set to produce an ERROR condition upon failure.

The heart of any checker is the `do_check:` attribute at the bottom. This is where the actual checking takes place. You can see here that there is a sequential series of steps that first collects a list of the unparameterized features:

```
$unparam_feat_list <<
    mqc_askFeatures( feature_type, BREP );
```

...and then creates a detail message based on the number of unparameterized features found:

```
$detail_msg << "Found "
    + Stringvalue( Length( $unparam_feat_list ) )
    + " unparameterized feature(s) in part.";
```

At the end, there is a conditional IF statement that says, “If there are more than zero unparameterized features OR if the log type is set to INFO, then log a message. Otherwise (ELSE) do nothing.”

```
If ( ( Length($unparam_feat_list )>0 ) | ( log_type:=3 ))
Then
    ug_mqc_log( Nth( log_type:, log_type_option: ),
        $unparam_feat_list,
        $usr_msg + $detail_msg )
Else
    donothing;
```

We will make a few simple changes to this checker to make it look for suppressed features instead of unparameterized features.

□ Edit the `do_check:` attribute as follows:

```
(Any Uncached) do_check:
@{
    $suppressed_feats << ug_askSuppressedFeatures();

    $detail_msg << "Found "
        + Stringvalue( Length( $suppressed_feats ) )
        + " suppressed feature(s) in part.";

    $usr_msg << If ( "" = log_msg: )
        Then "" Else log_msg: + "~n";

    If ( ( Length($suppressed_feats )>0 ) | ( log_type:=3 ))
    Then
```

```

        ug_mqc_log( Nth( log_type:, log_type_option: ),
        $suppressed_feats, $usr_msg + $detail_msg )
    Else
        donothing;
};

```

- Edit the documentation (comments at the top) as follows to reflect the purpose of the new checker:

Description:

This checker analyzes the current work part and checks for existence of suppressed features.

Parameters:

(Integer) log_type - 1 - LOG_ERROR,
 2 - LOG_WARNING,
 3 - LOG_INFO, or
 4 - LOG_NONE.
 Default is 1
 (to log an error)

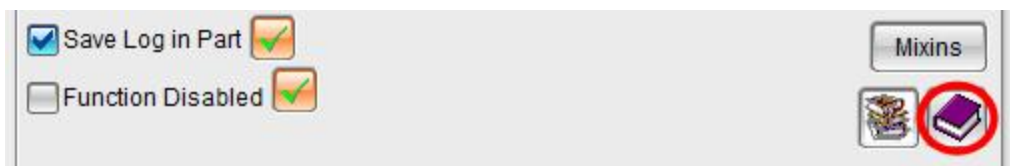
(String) message - message to be logged

Results:

PASS -- no suppressed features exist in
 the current part.

FAILED -- found suppressed features in
 the current part.

NOTE: This documentation can also be edited in the Check-Mate Author dialog before saving the check. This can be done using the **Documentation** button in the upper right area of the **Create / Edit Checker** tab of the **Check-Mate Author Tools** dialog.

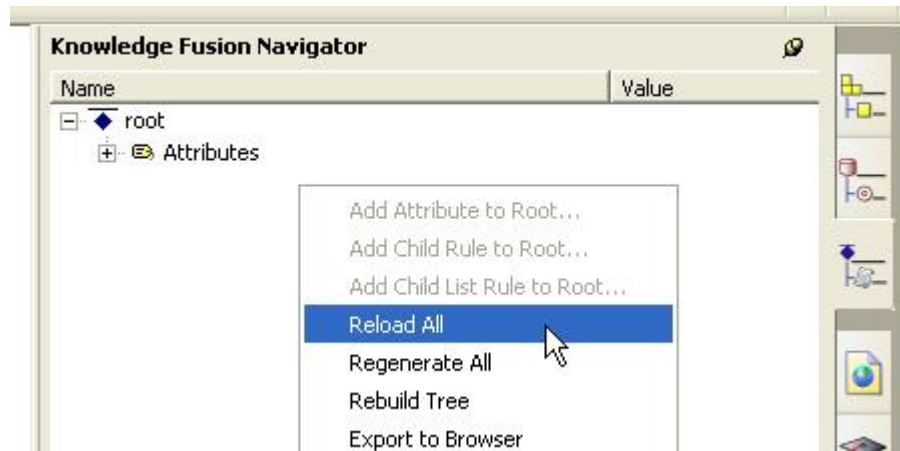


- Once these changes have been made, save the file.

Step 6: Reload all Knowledge Fusion code inside NX.

- Open the Knowledge Fusion navigator (on the right edge of the screen, under the Part Navigator. If the tab for this navigator is not visible, go to the **Start** button and choose All Applications → Knowledge Fusion in the drop-down menu to activate it.

- ☐ Right-click anywhere in the white space and choose **Reload All** as shown:



- ☐ The new check should now be visible in the tree in the Check-Mate user dialog.

Step 6: Test the new checker.

- ☐ Open the Check-Mate dialog and go to the “*Tests*” tab. Expand the **Check-Mate Training** category and find the new checker there.
- ☐ Create a test part containing a few modeling features.
- ☐ Test the checker against the valid features, and confirm that the part passes the checker successfully.
- ☐ Suppress one or more features and re-run the test. Confirm that the test fails appropriately.

[End of Activity]

Creating New Checks from Scratch

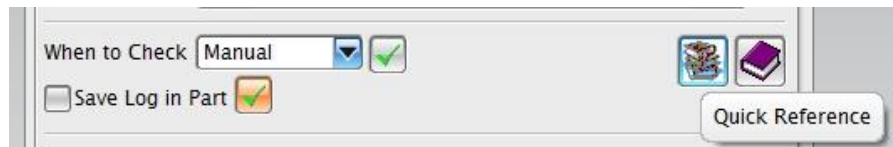
Frankly, creating a check completely from scratch will be extraordinarily rare. It has been said that the original “Hello World” program was the only original program written in C, and that every other C program written since was created by performing a “Save As...” operation from that original.

Check-Mate customization is very similar. Rarely (if ever) is a new check so radically different that it is not valuable to start with a “Save As...” of an existing check. Refer to the last activity for an example of this.

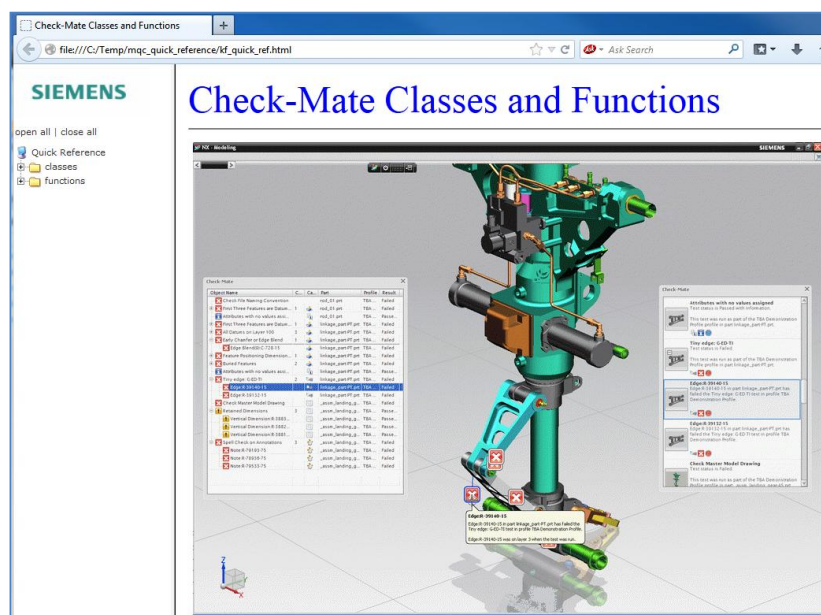
Finding What You Need (Where are the docs hiding?)

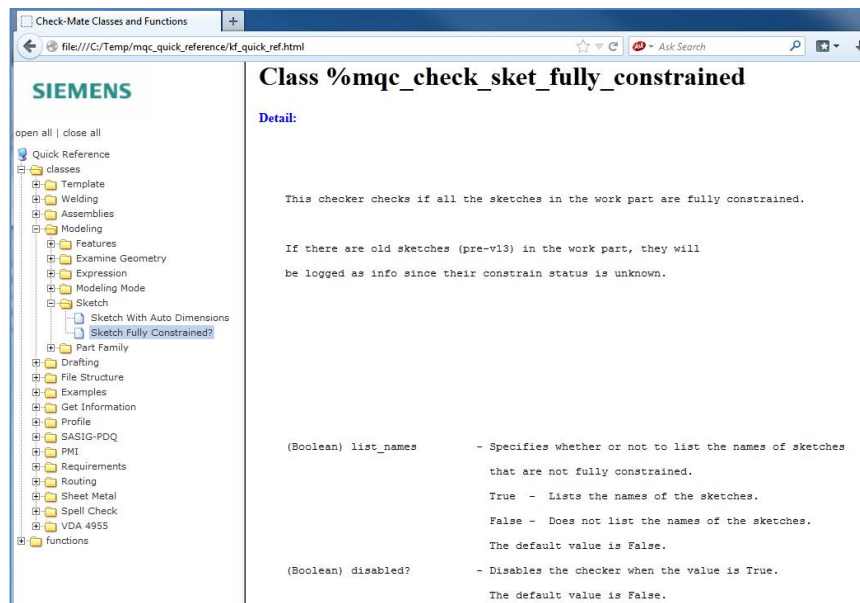
The main trick when modifying existing checks is finding new Check-Mate (and/or core Knowledge Fusion) functions that perform the tasks you need.

The very best source of these is the **Quick Reference** command inside the **Check-Mate Author Tools** dialog. This is near the top on each of the three tabs:



When pressed, this button creates (and launches in your default browser) an HTML file listing every available checker class and checker function currently loaded in your NX session. Because this list is dynamically generated, it takes a minute or so to create it the first time. You can then search through it.





(An example of a checker class)

A common workflow involves:

1. Searching first in the list of existing checkers to choose a checker that is at least close – one that either:
 - a. Deals with the same kind of entity being considered (Component, layer, solid, point, etc.) or...
 - b. Has a similar logical function (Look for all... Find any missing... Fail if any..., etc.).
2. Performing the “Save As...” operation
3. Searching the HTML reference for functions that either:
 - a. Provide direct access to the set of objects under consideration
 - b. Provide INDIRECT access to the set of objects needed
 - c. Filter or reduce a larger list of objects down to the needed set
4. Combining the functions in useful ways to collect or focus in on the requirement at hand.
5. If there is an error, report something useful to the user.

One Simple Example of “Connecting the Dots”

One example of this might be a hand-built checker that reports an error when it finds:

Any layers containing entities that are not part of a category.

Looking at this requirement, you might recognize that we provided a new checker recently to address this directly. We’ll ignore that fact for just a moment and look at the combination of functions that would help us address this requirement.

First of all, we might start with a list of all layers that are not currently in any category. Another name for these layers is “orphan layers”. The function `mqc_ask_orphan_layers()` will give us this list.

This is obviously a larger set of layers than we need, but we know that it includes the ones we need. There are likely a large number of empty layers included in this list as well. It’s not a bad starting point, though, as it’s at least smaller than the entire list of ALL layers.

To focus in on only the layers that contain entities, we might ask for a list of the entities on these orphan layers, and then ask those entities which layers they are on. There are two handy functions to do this:

`mqc_ask_layer_entities()` will ask for a list of all entities on our orphaned layers, and `mqc_collect_entity_layers()` will turn around and ask those entities which layers they came from. The result will be the list of layers we want – the orphaned layers (layers not part of a category) that contain entities.

This process of “connecting the dots” is the real challenge (and fun) of Check-Mate customization.

To help you quickly arrive at the set of objects you’re really looking for, there are many good “filtering” functions available.

`mqc_collectEntitiesWithFilterOptions()` and `mqc_selectEntitiesWithFilters()` are two of the most useful.

Functions like `mqc_ask_entities_by_type_name()` or `mqc_askEntities()` are also very handy for quickly collecting all entities that meet certain conditions.

Searching Tips

If when searching in the checker list or in the HTML documentation you do not immediately find the exact function you’re looking for, try again with slightly

abbreviated names. For instance, if you're searching for a particular function dealing with layers and categories, you might start by searching for "layer" and/or "category" and if those key words don't provide you with useful results, then trying "lay" and/or "cat" may return you a slightly larger set of results to try.

Also implied in the preceding paragraph is the concept that when searching for a Check-Mate function, you are almost always trying to combine two concepts to refine or expand a list. Layers and Categories were the example above. It may be a color and a layer, or an entity type and a density, or any number of things. Always remember that searching for both "ends" of the equation will often be helpful.

If you're really stumped and not finding anything that works, searching in the dfa source code in the UGCHECKMATE folder is sometimes a productive last resort. Searching for a text string embedded in any file named "*.dfa" will get you access to the comments and documentation embedded inside the dfa code as well. Hopefully you won't need to go that far.

Falling Back to the NX/Open API

As a last resort, KF can call functions written using the NX/Open API as well. This topic is beyond the scope of this course, but is not terribly difficult for a person who already knows both KF and the NX/Open API. This then provides tremendously deep and broad access to objects within NX.

Of course, as the Common API framework matures, functional coverage will become symmetric across all of the API languages available for NX, and the need to use another API will disappear.

Error Messages in Custom Checks

One last thing... When you create your own checks, take the time to provide the user with meaningful feedback that will help guide them toward how to identify and/or fix the problem. If your code causes a check to return an error for a specific reason, make your error message tell the user about that reason. The clearer your error messages are written, the quicker your users will be able to rectify the problem and move on.

Section 4: Knowledge Fusion

Before you can effectively customize Check-Mate much further you will need to become familiar with the Knowledge Fusion programming language. This is the language in which custom Check-Mate checks will be written. In this section we will discuss the basics of KF in general, later we will discuss Check-Mate specifics. The KF language is designed to enable very complex custom applications. Programming for Check-Mate is quite simple compared to what KF is capable of handling. Therefore, many aspects of KF will be only briefly mentioned or skipped altogether.

Overview

General Qualities

Those of you who are experienced programmers may find Knowledge Fusion somewhat unusual. The following are attributes of the KF language:

- High level language (it takes little code to get a great deal accomplished).
- Programs are interpreted as executed (as UNIX shells or DOS) rather than compiled (as C, C++ & Fortran).
- Object oriented. Code for a given application is accessed through a class.
- Declarative instead of procedural. This means code is not necessarily executed in the order it appears in the dfa file. NX determines the order of execution based on dependencies. In some cases you may be required to specifically specify the execution order using `demandOrder` and `demandValue` mechanisms.
- Embedded in the update cycle of a part. This means that objects created by KF are updated each time a part is modified and updated. Objects created by KF code are modified (vs. recreated) on subsequent update cycles. This means you do not have to worry about deleting old objects (even when the number of objects changes).
- Strongly integrated with UI Styler (NX dialog building application). Unlike typical C or C++ (NX/Open) programs which require lengthy, complicated and sometimes confusing callbacks to interface with interactive dialogs, KF code simply needs to use a few naming conventions. NX provides callback mechanisms which allow more detailed control of the dialog. These callback mechanisms are still simpler than those in other languages. KF also has classes to create BlockStyler-type UIs.
- Strongly typed, e.g.: a number is a number and cannot be stored in a string variable without the use of a conversion function.

Programming elements

KF programs are made of elements written (coded) in a DFA file (ASCII text file with a ".dfa" file extension). The KF program elements consist of:

- KF version
- Comments
- Attributes
- Classes
- Expressions
- Conditional Statements
- Child & Child List Rules
- Referencing
- Functions and Methods
- Loops
- Frames

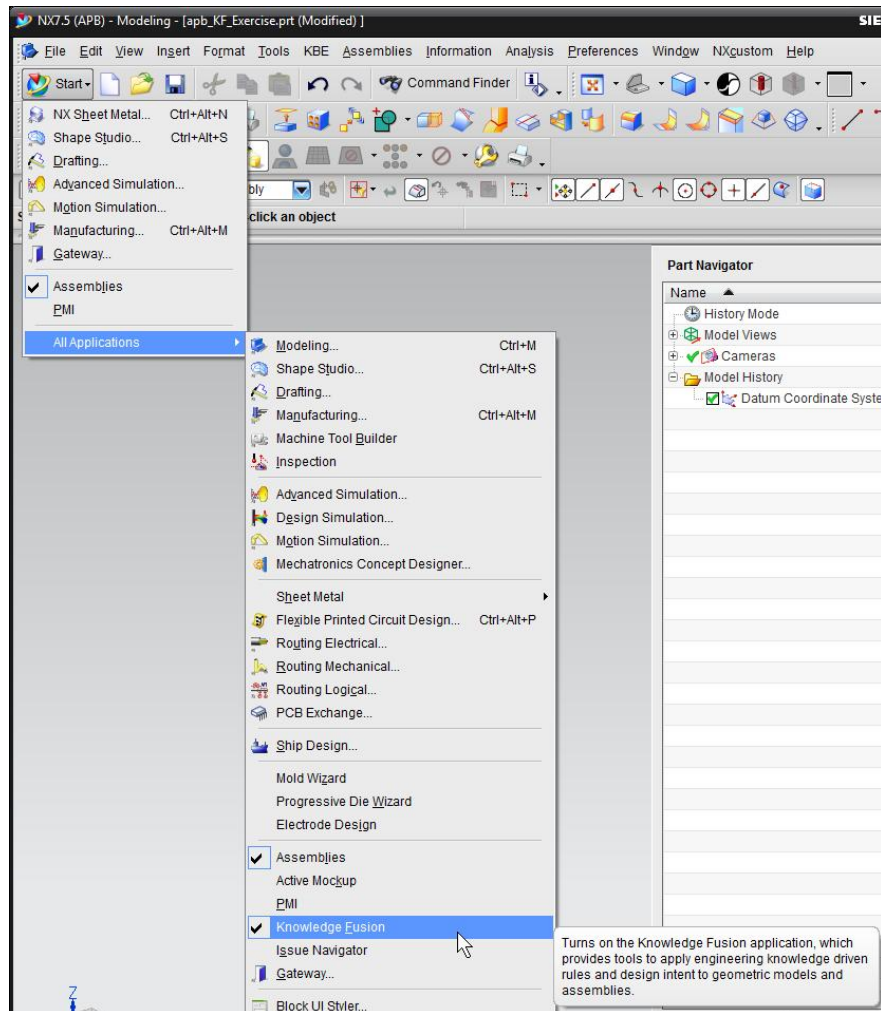
Activity 10 - How to Add Attributes

This exercise demonstrates how to add attributes using the KF Navigator.

Step 1

Create a new part and access the KF Navigator.

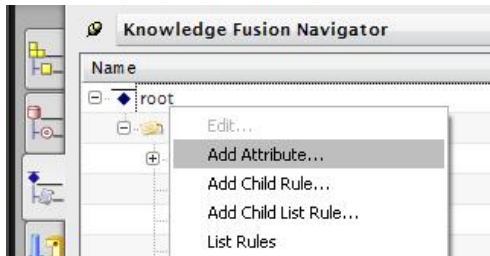
1. Choose File->New. Create the part file in a directory on the Desktop called KF. Enter ***_KF_Exercise in the File name box where *** represents your initials. Accept Units as metric. Choose OK.
2. Choose Start->All Applications->Knowledge Fusion.



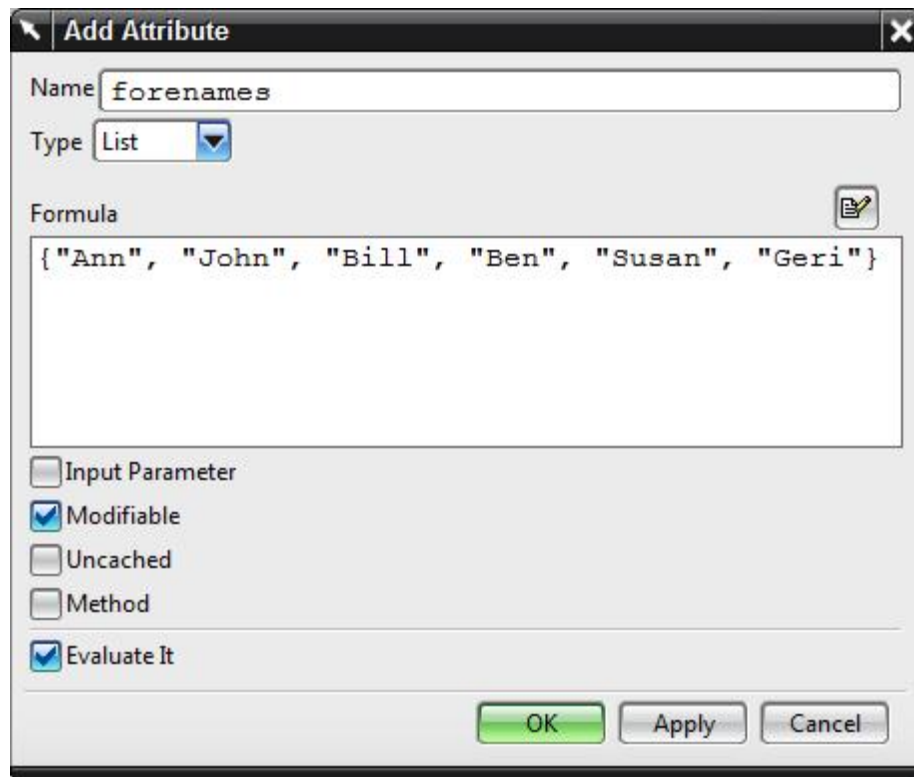
Step 2

Add a list attribute of forenames.

1. Hover the cursor over root and click MB3. Choose Add Attribute. The Add Attribute dialog displays.



2. Enter forenames in the Name box.
3. From the Type list, choose List.
4. In the Formula box, enter {"Ann", "John", "Bill", "Ben", "Susan", "Geri"}

A screenshot of the 'Add Attribute' dialog box. The 'Name' field contains 'forenames'. The 'Type' dropdown menu is set to 'List'. The 'Formula' text area contains the string {"Ann", "John", "Bill", "Ben", "Susan", "Geri"}. Below the text area are five checkboxes: 'Input Parameter' (unchecked), 'Modifiable' (checked), 'Uncached' (unchecked), 'Method' (unchecked), and 'Evaluate It' (checked). At the bottom are three buttons: 'OK' (highlighted in green), 'Apply', and 'Cancel'.

5. Choose Apply.

Step 3

Now using the same method create a list of surnames, as below: -

`{"Jones", "Smith", "Adams", "Brown", "Black", "Stevens"}`

Add Attribute

Name:

Type:

Formula:

☐ Input Parameter

☒ Modifiable

☐ Uncached

☐ Method

☒ Evaluate It

Step 4

Add more list attributes: -

1. Using the list data type, add a list attribute that will contain a list of lists called "master_data_list". The data should look like: -

```
{{"Ann Jones", 30, London},  
 {"John Smith", 41, Bristol},  
 {"Bill Adams", 25, Bristol},  
 {"Ben Brown", 50, Nottingham},  
 {"Susan Black", 30, Cheltenham},  
 {"Geri Stevens", 30, Preston}}
```

Add Attribute

Name:

Type:

Formula:

☐ Input Parameter

☒ Modifiable

☐ Uncached

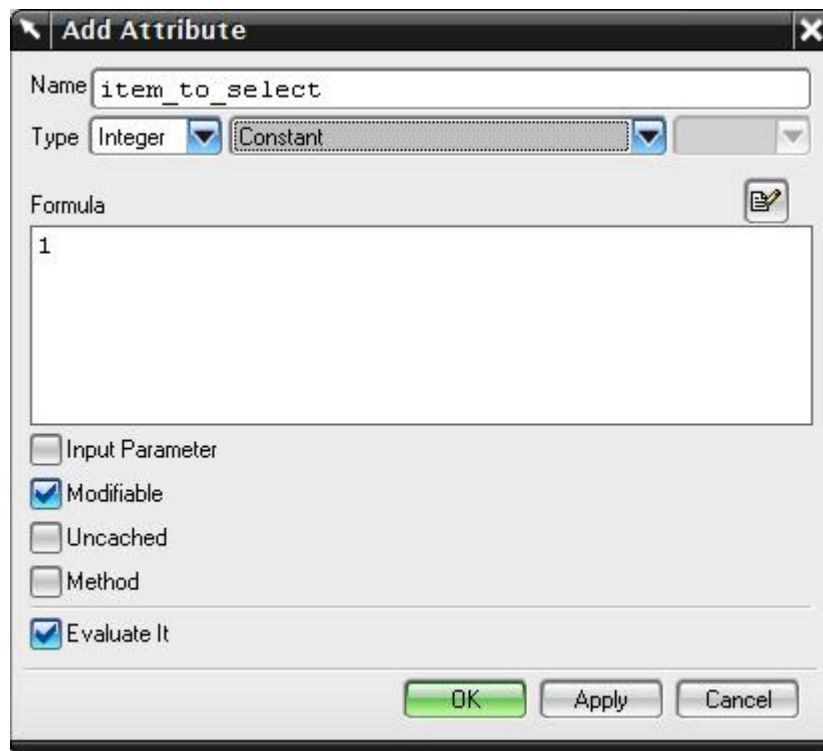
☐ Method

☒ Evaluate It

Step 5

Add an Integer attribute

1. Hover the cursor over root and click MB3. Choose Add Attribute. The Add Attribute dialog displays.
2. Enter item_to_select in the Name box.
3. From the Type list, choose Integer and Constant.
4. In the Formula box, enter 1.
5. Choose Apply



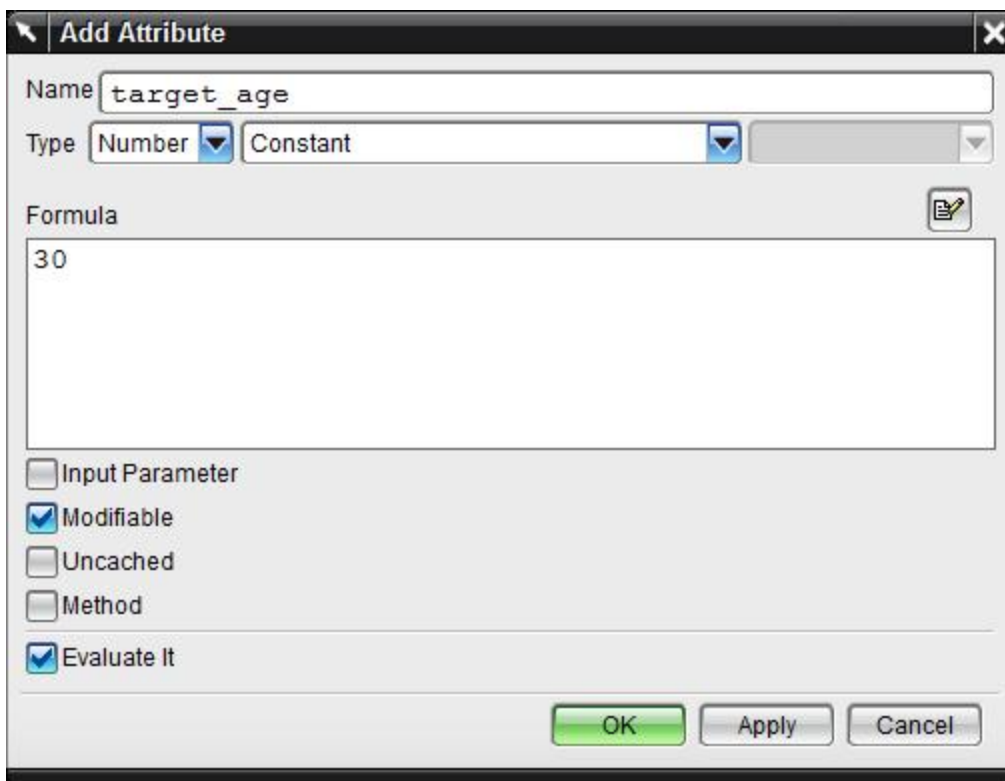
The screenshot shows the 'Add Attribute' dialog box with the following configuration:

- Name:** item_to_select
- Type:** Integer (selected in the dropdown)
- Constant:** Constant (selected in the dropdown)
- Formula:** 1
- Options:**
 - ☐ Input Parameter
 - ☒ Modifiable
 - ☐ Uncached
 - ☐ Method
 - ☒ Evaluate It
- Buttons:** OK, Apply, Cancel

Step 6

Add a Number attribute

1. Hover the cursor over root and click MB3. Choose Add Attribute. The Add Attribute dialog displays.
2. Enter target_age in the Name box.
3. From the Type list, choose Number and Constant.
4. In the Formula box, enter 30.
5. Choose Apply



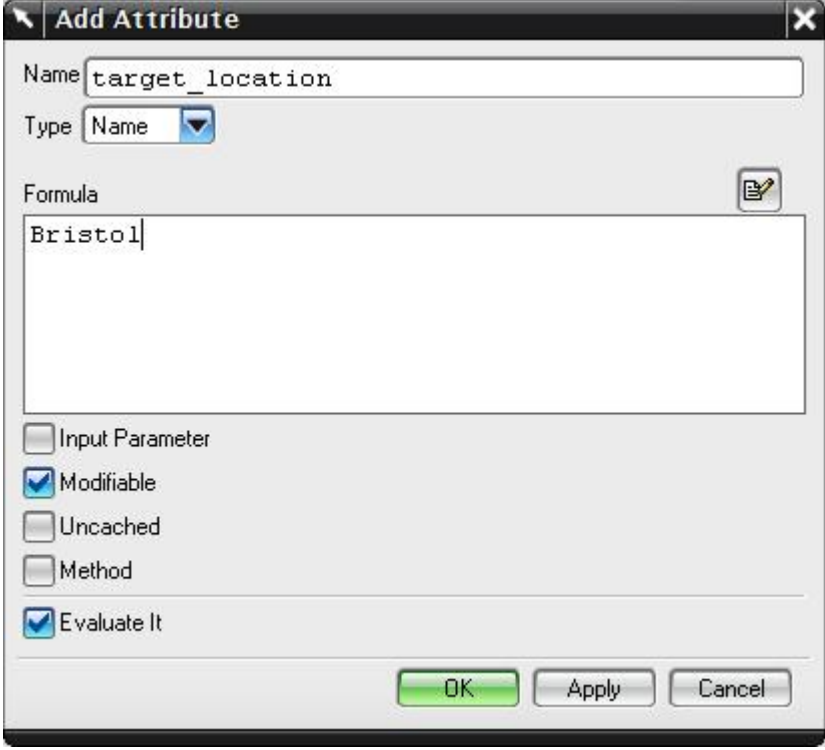
The image shows a software dialog box titled "Add Attribute". It contains the following fields and options:

- Name:** A text box containing "target_age".
- Type:** A dropdown menu with "Number" selected, and a secondary dropdown with "Constant" selected.
- Formula:** A large text area containing the number "30".
- Options:** A list of checkboxes:
 - ☐ Input Parameter
 - ☒ Modifiable
 - ☐ Uncached
 - ☐ Method
 - ☒ Evaluate It
- Buttons:** "OK", "Apply", and "Cancel" buttons at the bottom right.

Step 7

Add a Name attribute

1. Hover the cursor over root and click MB3. Choose Add Attribute. The Add Attribute dialog displays.
2. Enter target_location in the Name box.
3. From the Type list, choose Name.
4. In the Formula box, enter Bristol.
5. Choose Apply

The image shows a software dialog box titled "Add Attribute". It has a standard window frame with a title bar and a close button. Inside, there are several input fields and checkboxes. The "Name" field contains the text "target_location". The "Type" field is a dropdown menu currently showing "Name". Below these is a "Formula" field containing the text "Bristol". At the bottom, there are five checkboxes: "Input Parameter" (unchecked), "Modifiable" (checked), "Uncached" (unchecked), "Method" (unchecked), and "Evaluate It" (checked). At the very bottom of the dialog are three buttons: "OK", "Apply", and "Cancel".

Add Attribute

Name:

Type:

Formula:

☐ Input Parameter

☒ Modifiable

☐ Uncached

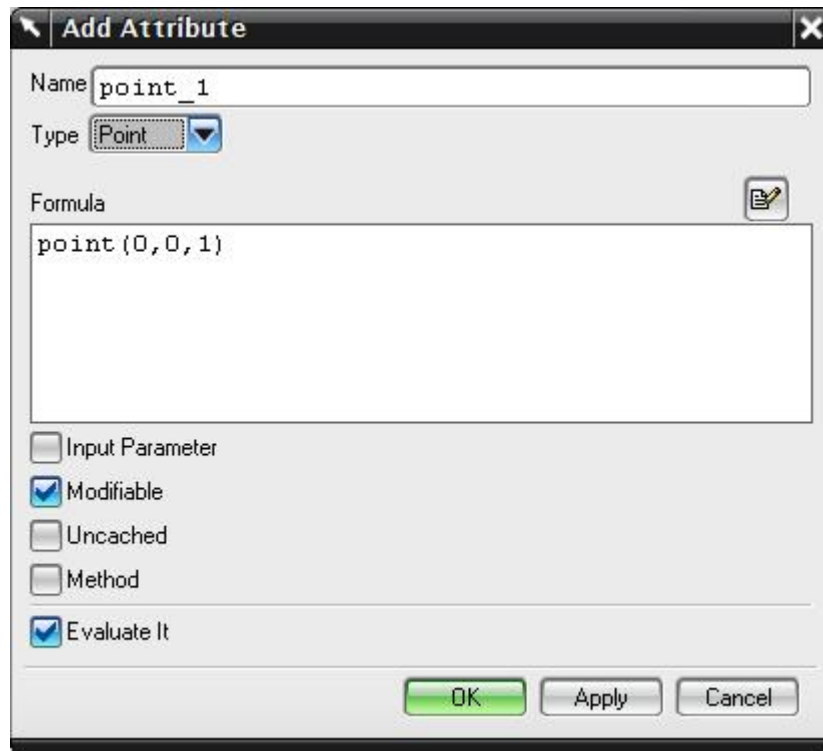
☐ Method

☒ Evaluate It

Step 8

Add an attribute of data type point.

1. Hover the cursor over root and click MB3. Choose Add Attribute. The Add Attribute dialog displays.
2. Enter point_1 in the Name box.
3. From the Type list, choose Point.
4. In the Formula box, enter point(0,0,1).
5. Choose Apply

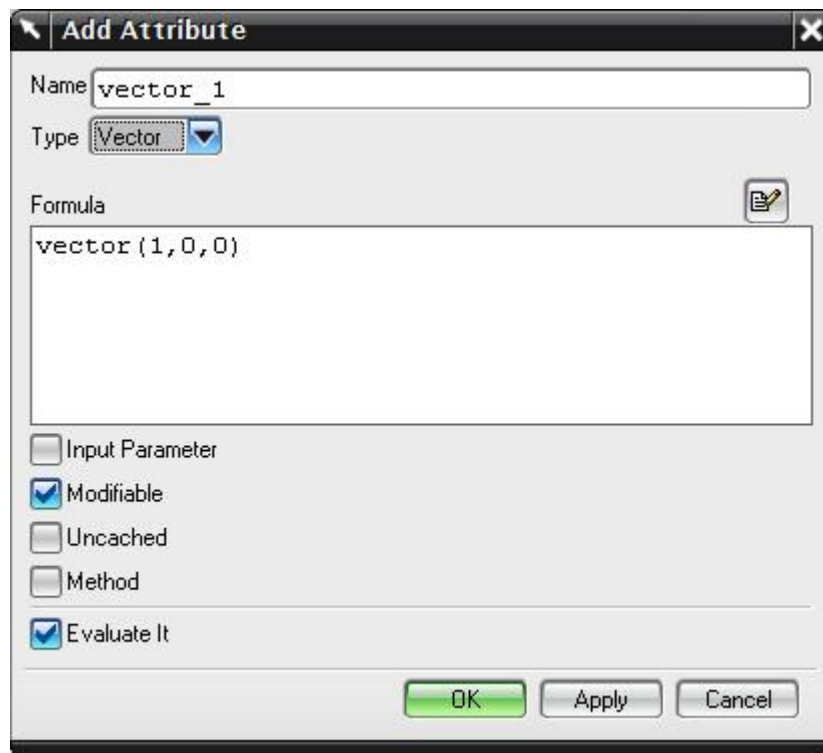


6. Now add a second point attribute called Point_2, with x,y and z co-ordinates of 100,1,0 respectfully.

Step 9

Add an attribute of data type vector.

1. Hover the cursor over root and click MB3. Choose Add Attribute. The Add Attribute dialog displays.
2. Enter vector_1 in the Name box.
3. From the Type list, choose Vector.
4. In the Formula box, enter vector(1,0,0).
5. Choose Apply

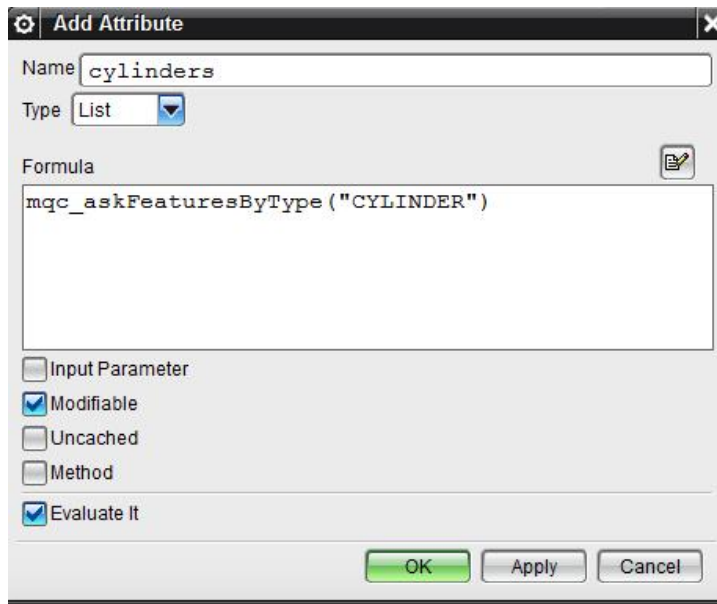


6. Now add a second vector attribute called vector_2, with x,y and z co-ordinates of 0,1,0 respectively.
7. Save the part.

Activity 11 - Defining Attributes Using Expressions

In this activity you will assign several expressions to attributes and view the results. To define each attribute, follow these steps:

1. In the KF Navigator right click on Root or Attributes and select Add Attribute...
2. Type the attribute name (without the :) into the Name Field.
3. Select the type (inside parentheses) from the Type dropdown menu.
4. Type the formula (everything following the :) into the Formula field.
5. For example: (list) cylinders: mqc_askFeaturesByType("CYLINDER");



6. Open the part CM_Expressions
7. In the KF Navigator, define the following attributes with expressions.
 - a. (list) cylinders: mqc_askFeaturesByType("CYLINDER");
 - b. (integer) num_cyl: length(cylinders:);
 - c. (string) cyl_string: format("There are %d cylinder features in the part", num_cyl:);
 - d. (list) primitives: loop {
 for \$type in {"CYLINDER", "BLOCK", "CONE", "SPHERE"};
 append mqc_askFeaturesByType(\$type);
};
 - e. (boolean) too_many?: if (length(primitives:) > 2) then true else false;
note you can simplify this to just length(primitives:) > 2 because this expression will evaluate to a boolean true or false.
 - f. (any) output: loop { for \$prim in primitives;; append { ug_printvalue(\$prim) };; }

[END of Activity]

Activity 12 - Loops

This exercise demonstrates how to create loops to iterate through list data.

Creating Loops

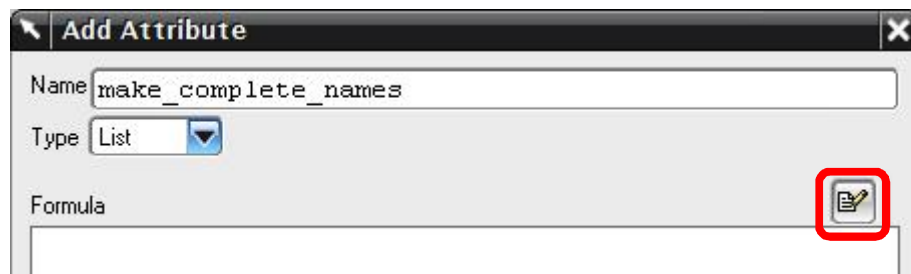
Step 1

Open the previously created part `***_KF_Exercise` and access the KF Navigator.

Step 2

Add a list attribute.

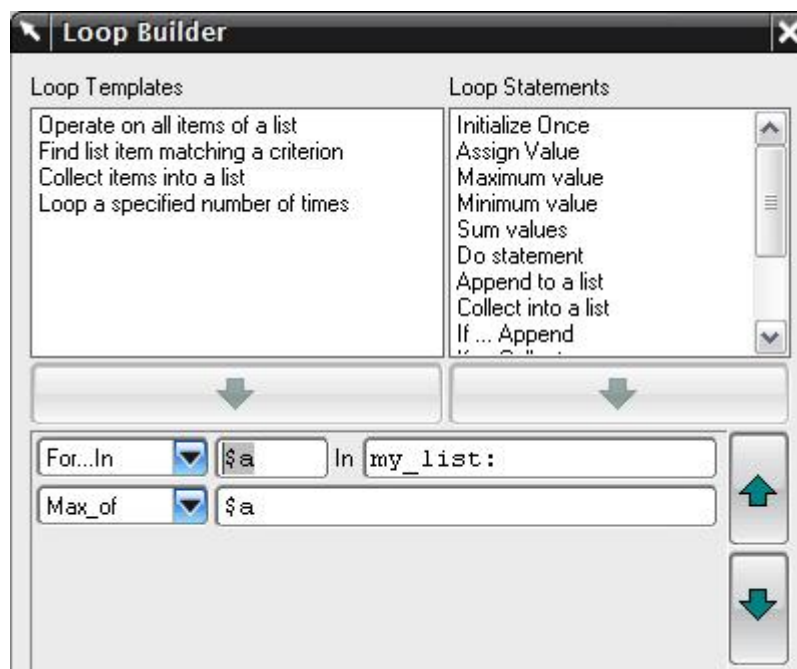
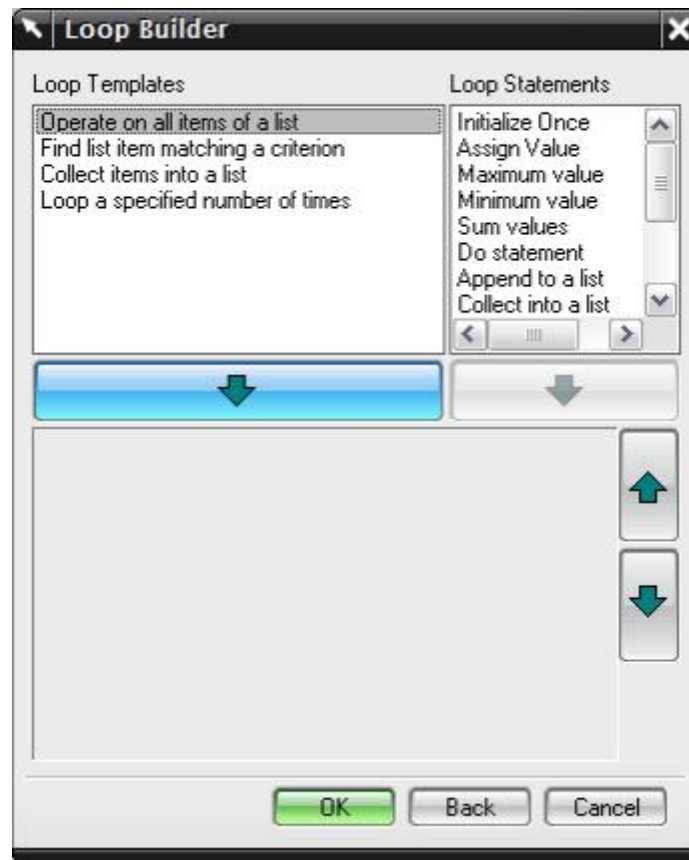
1. Hover the cursor over root and click MB3. Choose Add Attribute. The Add Attribute dialog displays.
2. Enter `make_complete_names` in the Name box.
3. From the Type list, choose List.
4. In the Formula box, select the extended text entry button to access the popup as below.



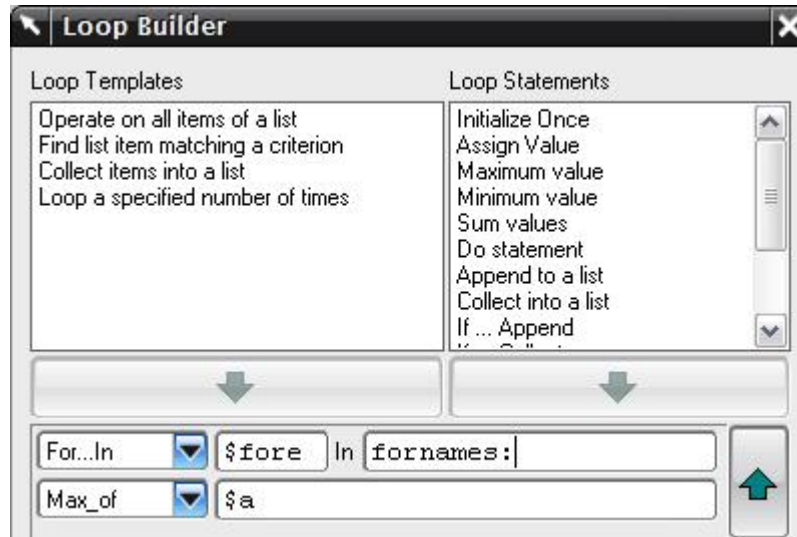
5. Select the Loop Builder icon 



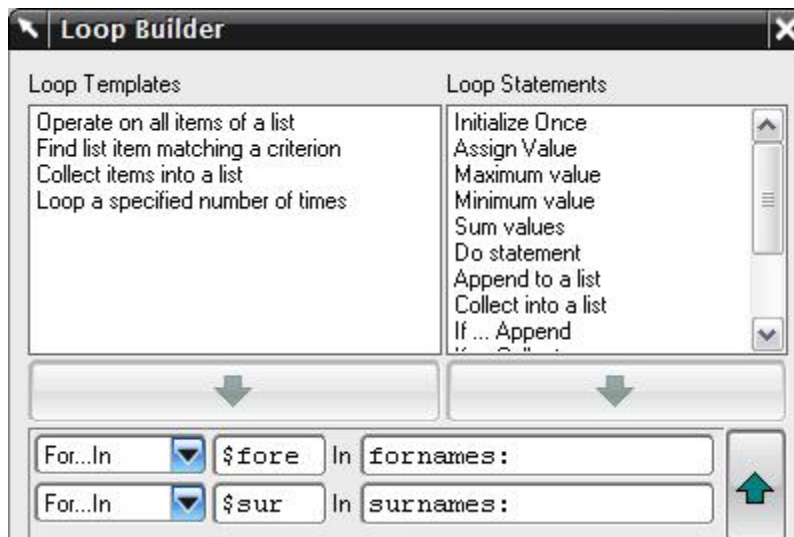
6. First click on “Operate on all items of list” and then the down arrow to Insert the Template.



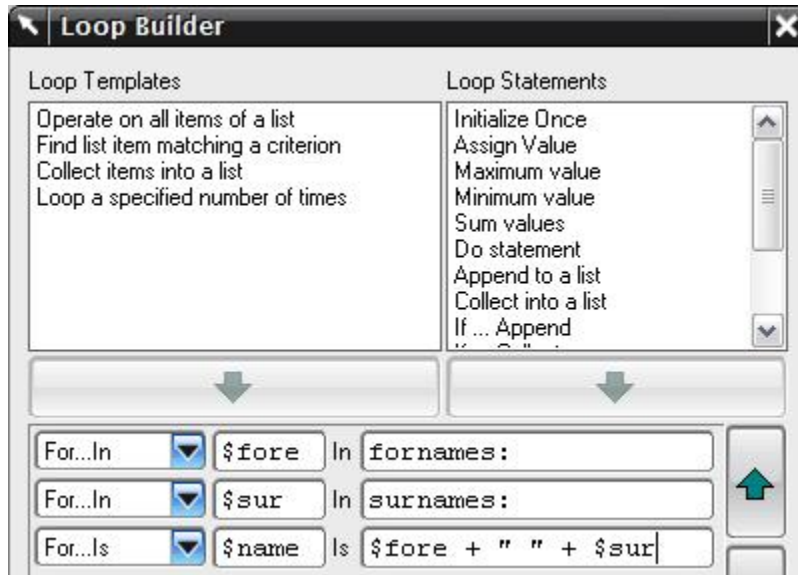
7. Modify \$a to read \$fore, and replace my_list: with a reference to forenames.



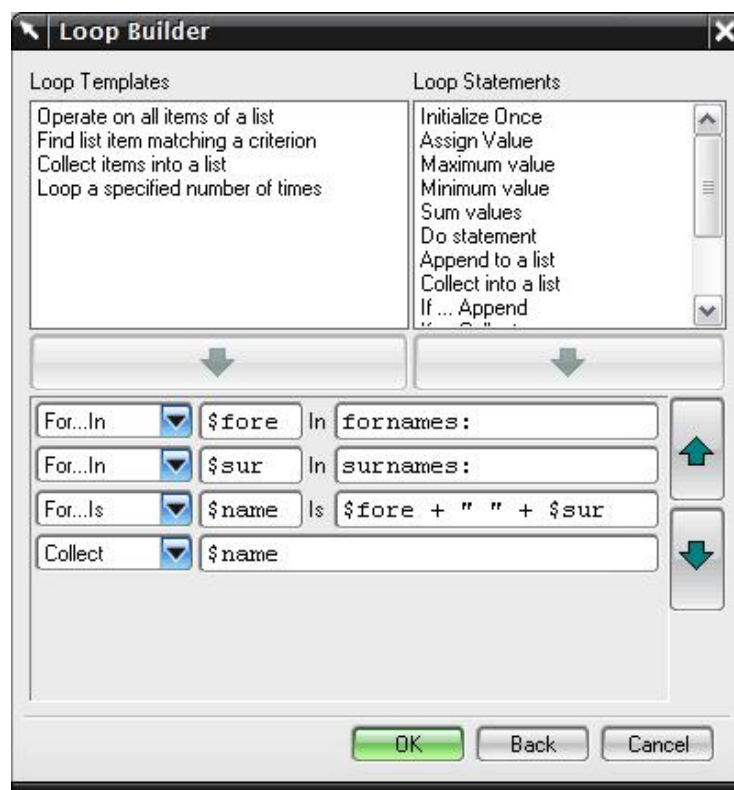
8. Now change the pull down option from "Max_of" to "For...In" and assign another temporary variable \$sur to reference elements of the surnames list attribute.

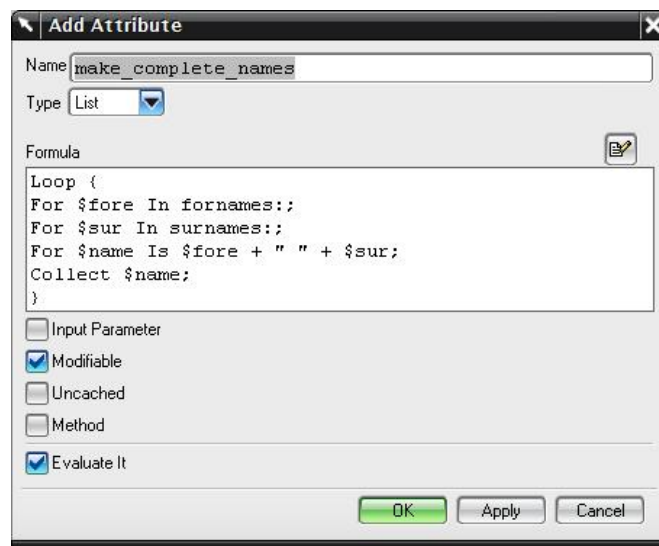
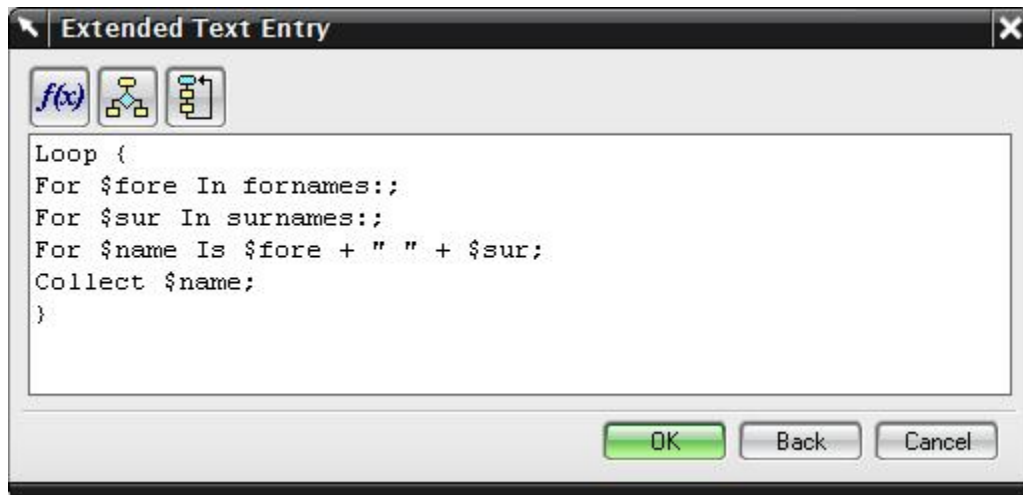


9. Now we need to select "Assign Value" from the Loop Statements options to add another variable line to our loop builder and create a variable called \$name as below. This new variable is to add together the strings returned by the variables \$fore & \$sur (also adding a space in between them) to create a new string.



10. Finally we now need to add a collector clause. Add another variable line and then change the pull down option to "Collect" as below. Then select "OK", and "OK" to return to the attribute constructor. Your loop should look like below.





11. When this attribute is evaluated, it will now step through both the forenames and surnames list and create a list of complete names.

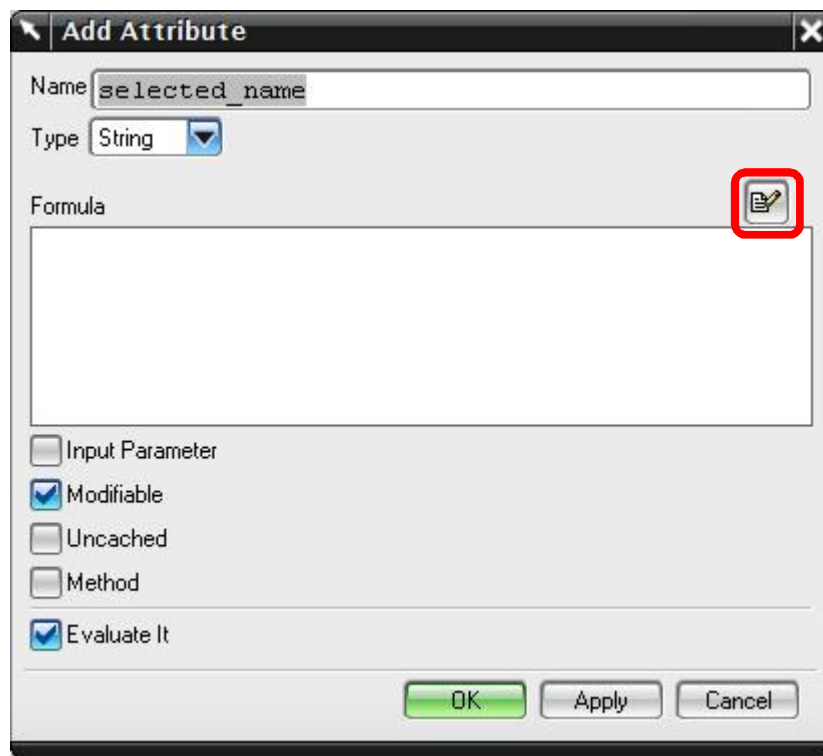


Step 3

The second loop we are going to build will step through the list attribute "master_data_list" searching for a target name and will then return the data against that name.

To accomplish this, first we need to identify a target name, we can do this easily by referring to a member of the "make_complete_names" list we created above. Do the following: -

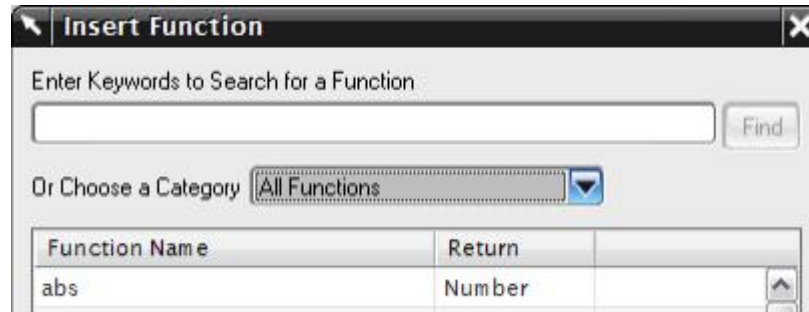
1. Make a new string attribute called "selected_name".
2. Select the extended text entry button .



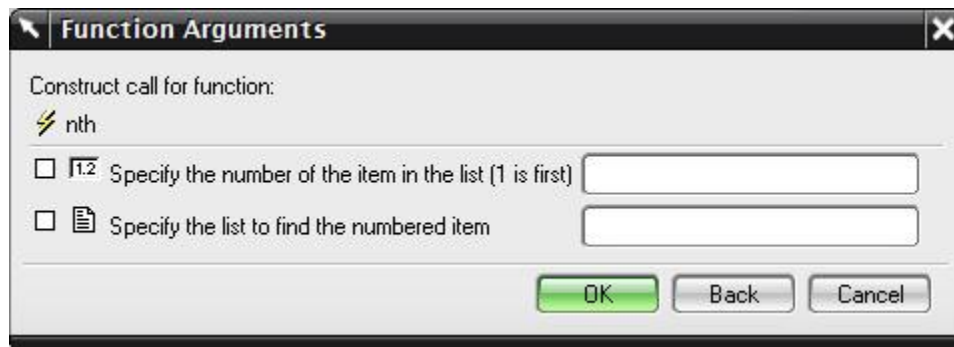
3. Now select the function button so that we can access the KF function libraries.



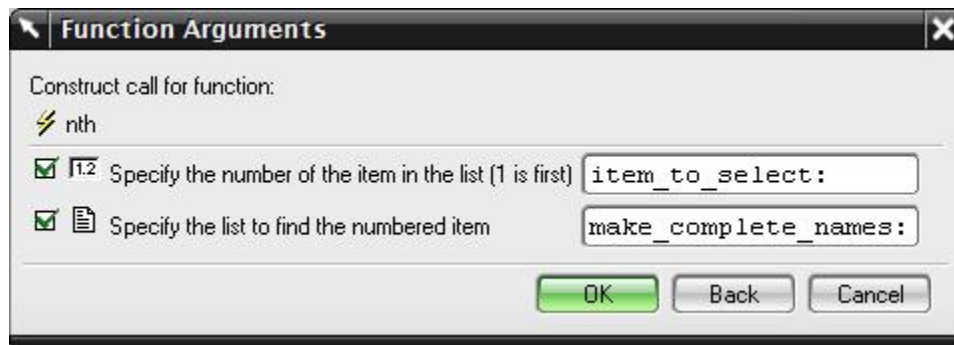
4. Within the function listing select "*All Functions*"



5. Scroll down and find the function "*nth*" and select OK. This function takes as arguments an integer that represents the position of the element we want to return and a list of data.
6. You will now see another popup that prompts you to supply the required inputs (or arguments) to the function, in this case an integer and a list.




7. Reference the attributes "*item_to_select*" and "*make_complete_names*" respectfully, select OK, OK. The attribute formula should now look as below. You should now be able to edit the value of "*item_to_select*" and see the value of this attribute update.



Add Attribute

Name

Type

Formula 

```
nth( item_to_select:, make_complete_names: )
```

☐ Input Parameter

☒ Modifiable

☐ Uncached

☐ Method

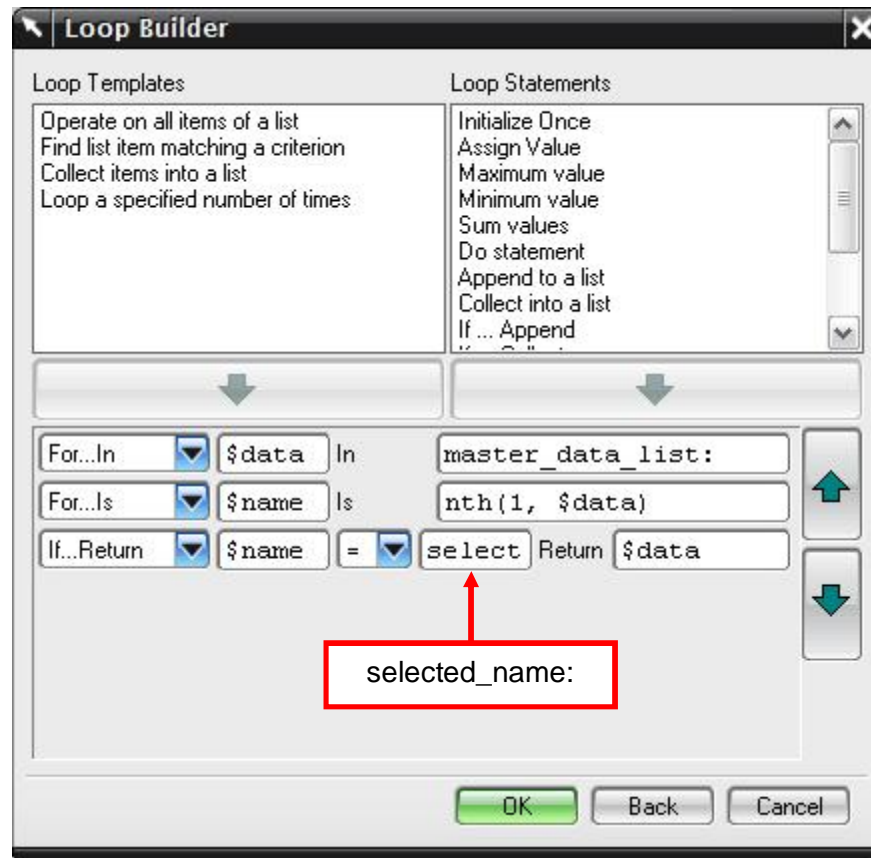
☒ Evaluate It

referenceFrame (Frame)	p(0, 0, 0), v(1, 0, 0), v(0, 1, 0)
saveClass? (Boolean)	
+ saveClassList (List)	
saveClassMixins? (Boolean)	
+ saveValue (List)	
selected_name (String)	"Ann Jones"
+ surnames (List)	List of 6
taget_age (Number [mm])	30
target_location (Name)	Bristol
vector_1 (Vector)	v(1, 0, 0)
vector_2 (Vector)	v(0, 1, 0)

Step 4

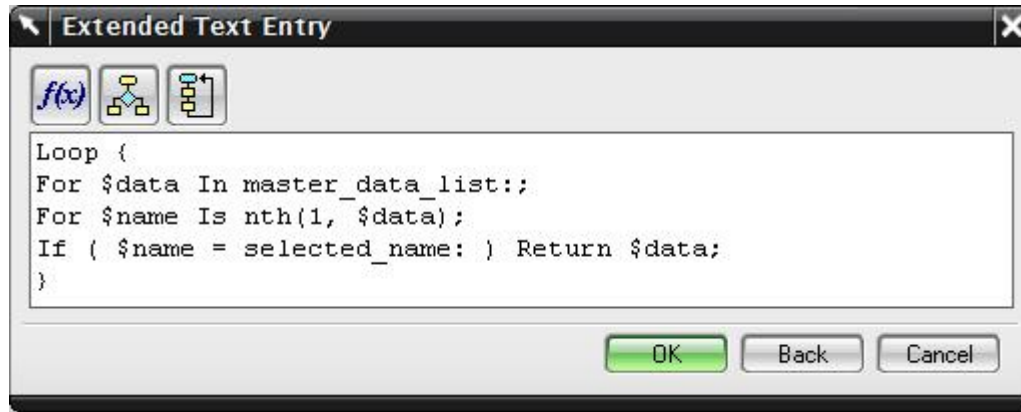
Now we have a target name, we can now declare another loop to use this target name to search within the "master_data_list" and return the data list that contains this name.

1. Create a new list attribute called "selected_record_by_target_name".
2. Use the same steps as earlier to enter the loop builder.
3. Set up your loop so it looks like below.




4. The loop above steps through our "master_data_list" attribute. Remember that this is a list of lists, so at each iteration the loop assigns a list to the temporary variable \$data. Since this is a list, we need to use the "nth" function again to access a value from the list \$data that represents the name. Finally we carry out a test to see if the name in the list matches the value of the attribute we created earlier called "selected_name"; if it does it returns the list \$data.

5. Note that since we used the keyword `return` within this loop it will terminate when it reaches the first value that passes this test.
6. Your final loop should look like below. You should now be able to edit the value of `"item_to_select"` and see that the value of this attribute changed accordingly.




selected_record_by_target_name (List)	List of 3
(String) "Ann Jones"	
(Integer) 30	
(Name) London	

7. Now create a new list type attribute called `"selected_record_by_age"` that will use the attribute `"target_age"` as criteria to retrieve data from the `"master_data_list"`. In this instance we want to return all the data entries that contain our target data, so try using `"If Collect"` in the collect clause rather than `"If ... Return"`. For a target value of 30 your return list should look like: -

 selected_record_by_age (List)	List of 3
⊖ (List) List of 3	
... (String) "Ann Jones"	
... (Integer) 30	
... (Name) London	
⊖ (List) List of 3	
... (String) "Susan Black"	
... (Integer) 30	
... (Name) Cheltenham	
⊖ (List) List of 3	
... (String) "Geri Stevens"	
... (Integer) 30	
... (Name) Preston	

8. Now create a new list type attribute called "selected_record_by_location" that will use the attribute "target_location" as criteria to retrieve data from the "master_data_list". In this instance we want to return all the data entries that contain our target data, so try using "*If Collect*" in the collect clause rather than "*If Return*". For a target value of Bristol your return value should look like: -

 selected_record_by_location (List)	List of 2
⊖ (List) List of 3	
... (String) "John Smith"	
... (Integer) 41	
... (Name) Bristol	
⊖ (List) List of 3	
... (String) "Bill Adams"	
... (Integer) 25	
... (Name) Bristol	

9. Finally test what happens when you set your target age and target locations to 70 and Glasgow respectively.
10. Save the Part

Activity 13 - Functions

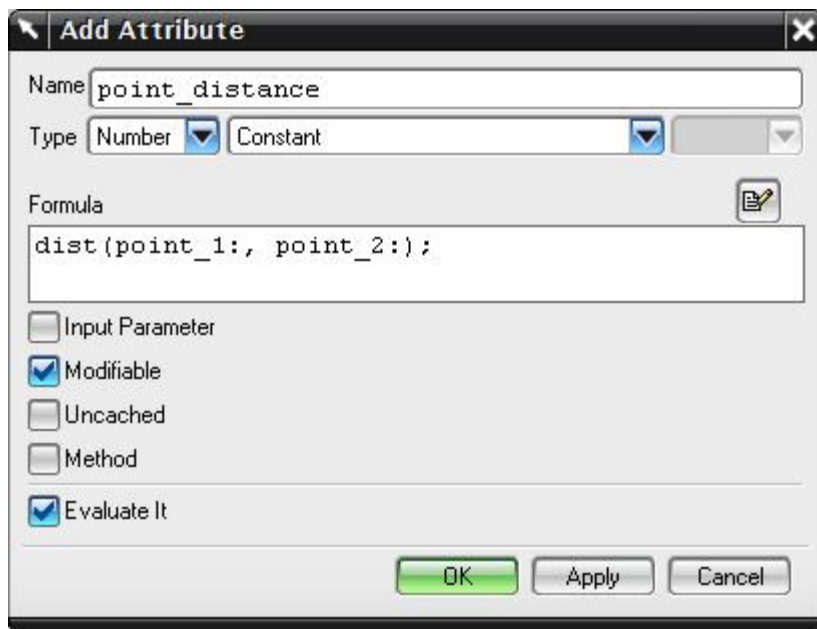
In the previous exercise we touched on calling functions by using the function “*nth*”.

Functions behave the same way as methods however they differ in that you can only access a method by referencing an object, whereas functions are independent of object (or class) definitions.

Within this exercise we will practice calling some pre-built functions and then move onto creating our own custom functions.

Step 1 – Calling Functions

1. Open the previously created part `***_KF_Exercise` and access the KF Navigator.
2. Create a number and constant type attribute called “point_distance”.
3. Navigate to the function library and find the function “dist”. This function measures 3D distance between two points (data type points not geometric entities).
4. This function requires two point arguments, so reference the previously created data type point attributes “point_1” & “point_2”.



5. Next we will create a Boolean type attribute that will call a function to measure the angle between two vectors and return FALSE if the angle is above zero, i.e. if the vectors are not in the same directions.
6. First create a Boolean type attribute called “vectors_same_direction”.
7. Navigate to the function library to find the function “Angle2Vectors”. You will see that this function takes as arguments 3 vectors; the first two vectors are the ones you wish to measure between, whilst the third vector is a reference vector about which the angle between the vectors is measured.
8. Reference vector attributes “vector_1” and “vector_2” and use the Z vector as the reference vector.
9. This function will return a number, however we want to add a test to see if vectors 1 and 2 are aligned; so add an “if” statement as below.

Add Attribute

Name

Type

Formula 

```
if Angle2Vectors( vector_1:, vector_2:, Vector(0,0,1) ) > 0  
then FALSE else TRUE
```

☐ Input Parameter

☒ Modifiable

☐ Uncached

☐ Method

☒ Evaluate It

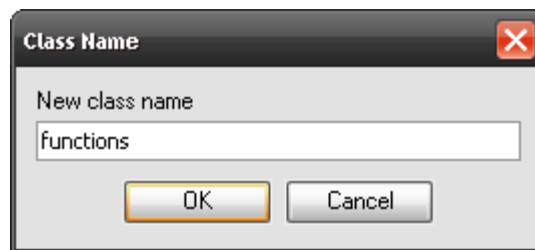
Step 2 – Creating functions

Functions have some things in common with methods;

- They can take arguments.
- They are defined as expression blocks.
- You have to declare the return type.

The easiest way for us to define our own functions is within a text editor (ICE, TextPad, Notepad). First we will create a function that will take as arguments two strings (a forename and a surname) and concatenates them together with a space between them to create a single complete name.

1. Create a new file, within ICE, File -> New Class..., name it functions. It will be saved where the dfa manager is currently set to point to; as only one directory is defined in the dfa manager that is where it will be saved.



2. Change the header, at the top of the file, to read `#! NX/KF 7.5` and `Defun: make_full_name (String $forename, String $surname)`

3. Then complete to create the following :

(The # comments are optional. They are designed to present additional information about the function when it is called.)

#! NX/KF 7.5

Defun: make_full_name (

#+

DesignLogic=no

Description:

#.Add the first and second string together with a space in the middle.#

#-

String \$forename, #. Input the first string to join.#

String \$surname #. Input the second string to join.#

)

@{

\$forename + " " + \$surname;

}; **String**

#+

Returns:

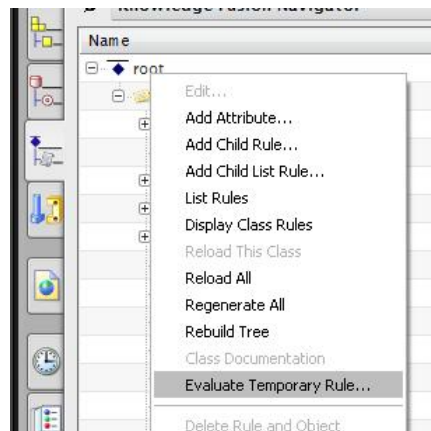
String - #.Returns a string joining both input strings together with.#

#-

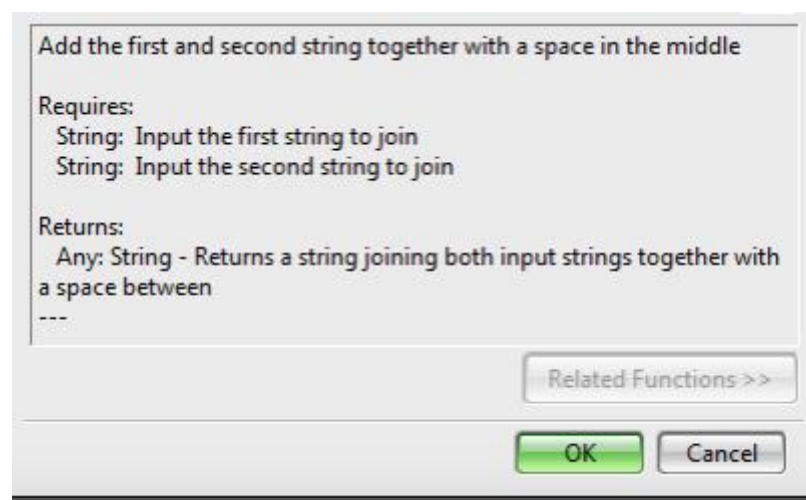
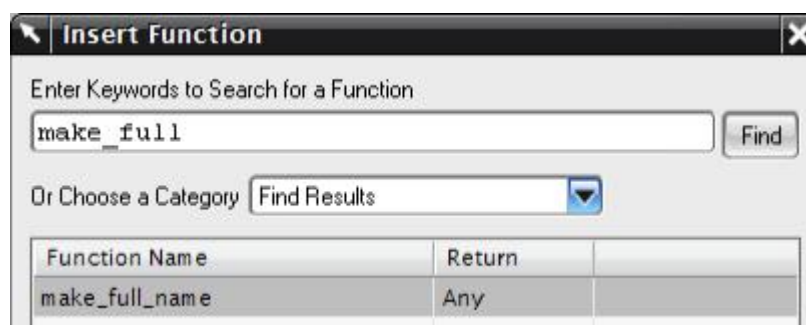
Save the dfa.

Now we can test our new function.

1. Right-click on the root in the KF Navigator and select "Evaluate Temporary Rule".



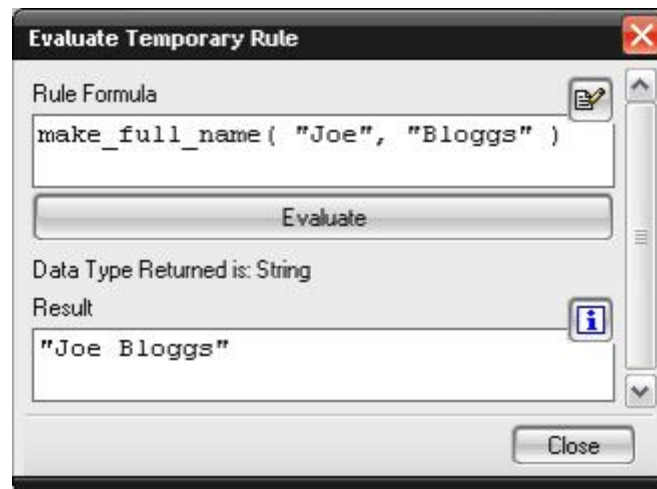
2. Navigate to the function library.
3. Search for the required function.



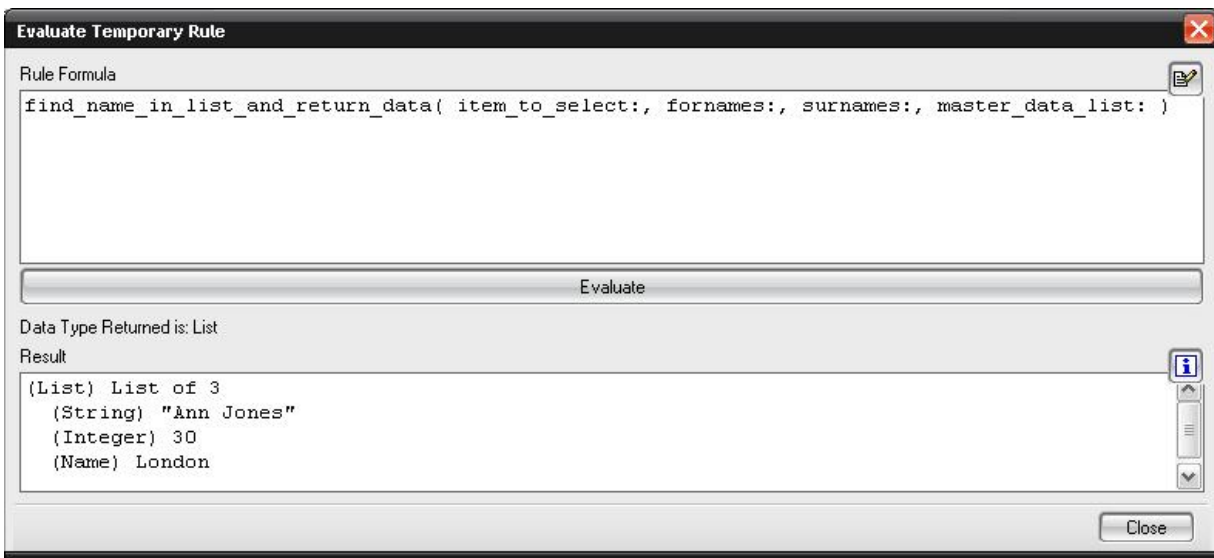
4. Notice the description information appears at the bottom of the dialog; this is because it was entered in the dfa file with the correct syntax.
5. Hit OK, and supply two string arguments to the function.



6. Hit OK and OK to return to the “Evaluate Temporary Rule” window, and then hit Evaluate; the returned value will be displayed in the lower window.



7. Now create a function called “find_name_in_list_and_return_data” that will take the following arguments
 - a) An integer that represents the position of the item we wish to reference in a list of forenames.
 - b) A list of forenames
 - c) A list of surnames
 - d) A master list of data that is defined as a list of lists.
8. Now define the function so that it: -
 - a) Uses the integer argument to retrieve forename and surname values.
 - b) Use our “make_full_name” function to concatenate these results together.
 - c) Define a loop that will use this data to retrieve the appropriate data from the master data list.
9. When complete create a new *list* attribute in our open part called find data using function, or use a Temporary Rule, and pass the required arguments to your function. This function should now return the same data that we previously returned using first a loop and then a method.



10. Finally we should enhance this function so that it will not error if either the item number supplied is incorrect or if it cannot find any records based upon the search criteria.
 - a) Add a test to your function that tests if the `item_number` supplied is greater than the number of items in the forenames list (see function "length").
 - b) Add a final test that evaluates the return value from a) and also check that some data has been found. If both are OK the function will return the data it has found, otherwise it will return an empty list, i.e. `{}`.
11. Finally, create a function that will take the returned data and format it together into a single string. To accomplish this you need to declare a function that takes a single list argument. Look at the function "stringvalue" to see how to change a value into a string.

```

Defun: find_name_in_list_and_return_data(
#+
DesignLogic=no
-----
Description:
#.
Function to find a name in a list and return the remaining data in that list
.#
-----
#-
integer $nth, #. nth in list to check the forename.#
list $forenames, #. a list of forenames to check.#
list $surnames, #. a list of surnames to check.#
list $master_list #. master list of list containing master data.#
)
@{
$test_nth << if length($forenames) < $nth then length($forenames) else $nth;
$forename << nth($test_nth, $forenames);
$surname << nth($test_nth, $surnames);
$names_joined << make_full_name($forename, $surname);
$find_data << loop{
    for $sub_list in $master_list;
    if (first($sub_list) = $names_joined) return $sub_list;
};
$result << if $nth = $test_nth then $find_data else {};
};list
#+
-----
Returns:
Number - #.Returns a list of data related to the search name,
if incorrect data is input returns an empty list.#
-----
#-

```

Section 5: Check-Mate Specific KF Elements


Anatomy of a Check-Mate Check

A Check-Mate checker uses standard Knowledge Fusion syntax with some specific required attributes and some mandatory organizational parameters. The diagram below shows the basic checker format found in every Check-Mate checker:

#! NX/KF 7.0	#1
DefClass: %cm_checker_anatomy (%ug_base_checker);	#2
<pre> ### <Description> Detailed description of what the class is checking </Description> <Parameters> List of parameters that the user can change (what and why) </Parameters> <Results> What the results mean (Pass/fail) and how to fix the problem </Results> #- </pre>	#3
# Required and common attributes	
(string) %test_category: "Check-Mate Training.Anatomy";	#4
(String) %displayed_name: "Check Anatomy";	#5
(Boolean Parameter) Disabled?: false;	#6
(Boolean Parameter) save_log_in_part: true;	#7
# Parameters that users can change while running checks	
(integer parameter) log_type: 1;	
(list) log_type_option: { LOG_ERROR, LOG_WARNING, LOG_INFO };	#8
(string) log_type_label: "Log Option";	
(string parameter) log_msg: "";	#9
(string) log_msg_label: "Log Message";	
# Link to documentation when right click on checker in results dialog	
(list) %do_doc: {URL, "www.mybusiness.com" };	#10
# Check-specific parameters and attributes	
(Number Parameter) tolerance: 0.001;	#11
# The do_check: attribute is always demanded when Check-Mate is run	
(any uncached) do_check:	#12

```
@{
    ...
    if (checkFailed) then
    @{
        ug_mqc_log(log_type:, $faliedObjectTags, $detail_msg);
    } else
    @{
        donothing;
    };
};
```

1. The first line: **#! UGNX/KF** is required as in all Knowledge Fusion code and specifies the version of KF that will be used to execute the code. This KF version number must always be on the first line of the file.
2. The **DefClass:** line defines the name of the checker you are defining. The name should begin with a % character to ensure that the class does not appear in the list when users are adding a child rule from the KF Navigator. Remember that all class names must be unique in the installation. The **%ug_base_checker** mixin must always be used for the class to be automatically included in the Check-Mate UI.

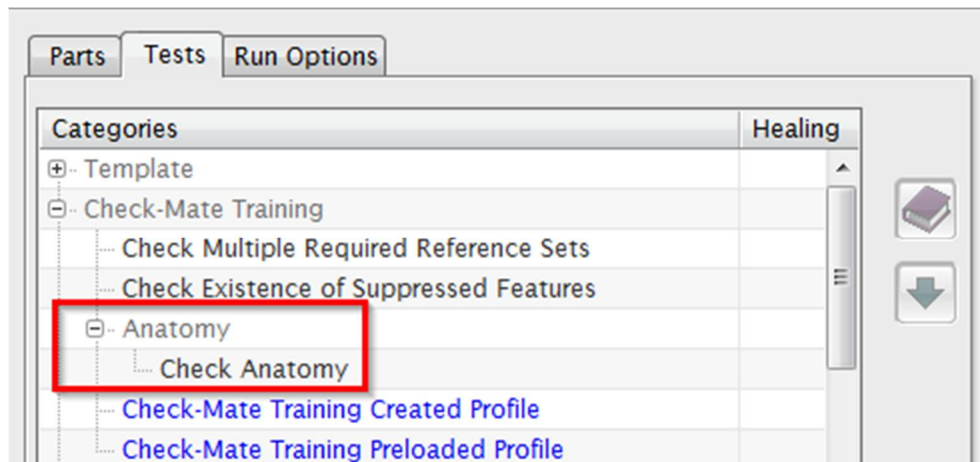
3. The area between **#++** and **#-** is for checker documentation. This documentation will appear when the user clicks the Documentation button  while the check is selected.

When creating your own checkers:


- a. Be as detailed as possible in these sections.
- b. Describe how to fix the problem found by the check.

When using the new automatic HTML generation utility, the Pseudo-XML tags here are also used to extract the appropriate text into the appropriate sections of the generated HTML file.

4. The **%test_category:** is a required parameter which specifies where the checker appears in the Categories list. Spaces are allowed in this attribute. The category can be made hierarchical by using a period (.) to separate levels. In the example above, the string: “**Check-Mate Training.Anatomy**” would put the checker at this position in the hierarchy:



5. The **%displayed_name:** is a required parameter which specifies the name for this checker that will appear in the Categories list. It is meant to be human readable and informative. Spaces are allowed in the displayed name.

6. The **Disabled?:** parameter indicates whether the checker is disabled or not. This can be determined by the user as the checker is selected by clicking the Customize  icon. Setting the value to false (as in the example) indicates that by default this checker will not be disabled. If this parameter is eliminated (or commented out), the user will not be allowed to disable this checker, because the Disabled toggle will not appear on the Customization dialog.
7. Similar to the **Disabled?:** parameter above, the **save_log_in_part:** parameter determines if the log results of the checker will be stored in the part file. Again, the value for this becomes the default and if this parameter is eliminated, then the user will not be able to change whether the log is saved in the part file or not.
8. The **log_type:** attributes are used to define the level of severity that will be reported if the check fails. Somewhere in the **do_check:** attribute (see 10 below) there must be a call to a function that reports errors to Check-Mate (usually **ug_mqc_log**), this error level is passed as a parameter to the function. You can limit the severity that a check can report by modifying the **log_type** attributes, eliminating some options or forcing a certain value.
9. The **log_msg:** attribute allows users to add their own informative text message to display if the check fails.
10. The **%do_doc:** list attribute allows the user to view external documentation. When right clicking on the check name in the Validation Results the indicated file will be opened. The list has two elements in the form { **NAME, STRING** }, where NAME can be any one of the following keywords:
- a. **HTML** - The appropriate browser starts to present the web page designated by the URL address in STRING.
 - b. **FILE** - The Information window opens to display the contents of the file specified by the file specification contained in STRING.
 - c. **TEXT** - The Information window opens to display the text contained in STRING.

Starting with NX 7.0 you will not need to set the **%do_doc:** attribute if:

- a. You are using HTML for your documentation **and**
- b. You put the HTML file in the same folder as the checker DFA file **and**
- c. You use the same base filename for both the DFA file and the HTML file (i.e.: my_new_checker.dfa and my_new_checker.html)

As long as these three conditions are met, then Check-Mate will automatically recognize the HTML documentation even if you have not specified a value for the **%do_doc:** attribute.


11. You can have a number of attributes that are specifically related to your check. Any attribute with the **parameter** behavioral flag on the data type declaration will appear as a modifiable item on the Customization dialog. Using this mechanism provides the ability for you to allow the end user to configure your check. You will learn more about this later.

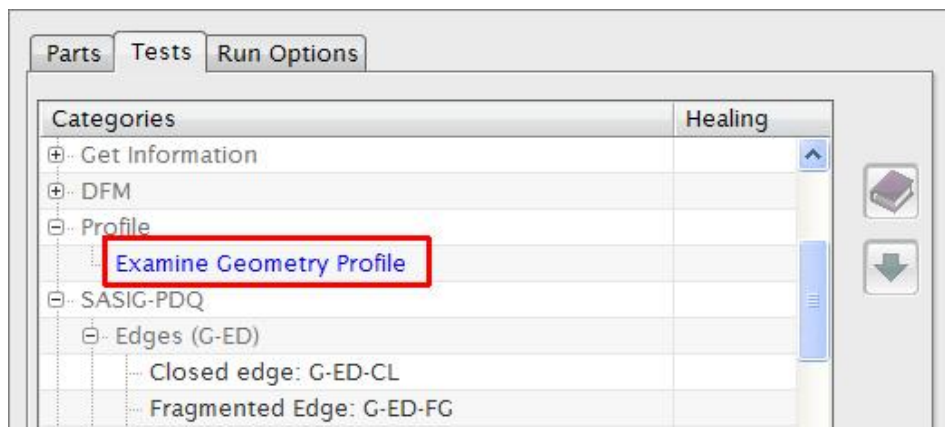
12. The **do_check:** attribute is required for all checkers. This is the one attribute that is always evaluated by Check-Mate. This is where the code makes the determination of whether the check passes or fails and if the failure should be information only, a warning or an error. The **do_check:** attribute:
- a. Will be evaluated every time the check is run.
 - b. Must end with a call to **ug_mqc_log()** (or a function that calls this) to send the results to the Check-Mate log.
 - c. Should always be **uncached** so that old results are not stored.

Anatomy of a Check-Mate Profile

A Check-Mate profile class uses an almost identical structure, with a few key differences:

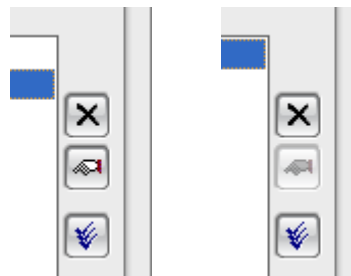
#! NX/KF 7.0	#1
DefClass: %my_profile_class (%ug_base_checker_profile);	#2
<pre> ### <Description> Detailed description of what the profile is checking </Description> <Parameters> List of parameters that the user can change (what and why) </Parameters> <Results> What the results mean (Pass/fail) and how to fix the problem </Results> #- # Required and common attributes </pre>	#3
(String) %test_category: "Get Information.PMI.GDT";	#4
(String) %displayed_name: "Check Name";	#5
(Boolean Parameter) Disabled?: false;	#6
(Boolean Parameter) save_log_in_part: true;	#7
(Boolean) allow_author_changes: false;	#8
<pre> # When will this profile be invoked? # 1 = manually, 2 = at update, 3 = at save time (Integer) check_time_index: 1; </pre>	#9
<pre> # Profile-specific parameters and attributes </pre>	
(Integer) an_arbitrary_attribute: 34;	#10
(Number Parameter) an_interesting_parameter: 12;	
<pre> # Checkers inside this profile </pre>	
<pre> (Child) %checker_name_1: { Class; %cm_train_01_check_class_name; }; (Child) %checker_name_2: { Class; %cm_train_02_another_check_class_name; }; (Child) %checker_name_3: { Class; %cm_train_03_yet_another_class_name; Tolerance; 0.01; Filter_parameter; {SOLID_BODIES}; }; </pre>	#11

1. The first line of a profile will be identical to a checker (or any other Knowledge Fusion DFA file.) This Knowledge Fusion version number must always be on the first line of the file.
2. The mixin for a profile is **%ug_base_checker_profile**, as opposed to just **%ug_base_checker**.
3. The area between **#++** and **#-** is for checker documentation. This section is  also identical to the checker description above.
4. The **%test_category:** attribute is the same as above.
5. The **%displayed_name:** attribute is the same as above. In the Check-Mate “Set Up Tests” dialog, profiles will appear in blue (whereas individual checks appear in black.)



6. The **Disabled?:** attribute is the same as above.
7. The **save_log_in_part:** attribute is the same as above.

8. The **allow_author_changes:** attribute not the profile parameters can be edited by Effectively, it turns the “customize” button (*Chosen Tests* list) on or off for a specific This is a simple but effective way to prevent changing settings, tolerance, or other default company’s standard profiles.



controls whether or the user at runtime. (to the right of the profile, as shown. users from values in your

9. The **check_time_index:** attribute indicates to Check-Mate when the profile should be executed – particularly in conjunction with automatic execution. By default (or if not specified) this attribute will have a value of “1”, which corresponds to manual execution from the “Execute Check-Mate” button in a dialog or from a toolbar button.

If this value has been set to “2” and this profile is added to the *Chosen Tests* list, then Check-Mate will execute this profile after every model update.

If this value has been set to “3” and this profile is added to the *Chosen Tests* list, then Check-

Mate will execute this profile as soon as the “Save Part” command is initiated. (In terms of sequence, the part will be checked first, and then saved.)


The “save time” option (“3”) is extremely handy for enforcing execution of Check-Mate across a set of users. That said, care should be taken to ensure that the specific checks executed in this automatic way do not take an excessively long time to complete. Some customers choose to include assembly integration formatting checks (which are generally very fast) at save time, and ask users to execute more heavy geometry integrity checks manually as needed, for instance.

10. Additional attributes, parameters and methods can be included in a profile class as needed. A few examples of this will be discussed in more detail as part of the section covering “Master Profiles” (a.k.a. “Smart Profiles” or “Conditional Profiles”) later in the course.
11. Finally, the checkers included in the profile are listed as individual **child** rules. In terms of what is actually happening here at runtime, NX is temporarily creating an instance of each of these checker classes.

If all necessary parameters have been pre-defined in the original checker (for instance, in cases where all of the out-of-the-box default values are acceptable) then the child class line will look similar to the first two examples above – the class name of each checker will be the only real parameter.

When you want to specify new values for input parameters for a checker, then the attribute-value pairs should be listed as shown in the third example above.

Checker Customized Dialog Details

The options that the user can access by clicking the **Customize**  icon for a checker can be fully defined in the KF code. The attribute type determines the type of widget created in the **Customize** dialog box. Labels, visibility, sensitivity, ranges and selection lists are all specified by carefully naming attributes. In order for the attribute to create a dialog widget, it must have the parameter qualifier (e.g. (string parameter) test_string: "Abc";).

The following extensions are recognized by the Check-Mate framework to indicate certain user interface styles and values to be used in parameter presentation/choices during “customization”:

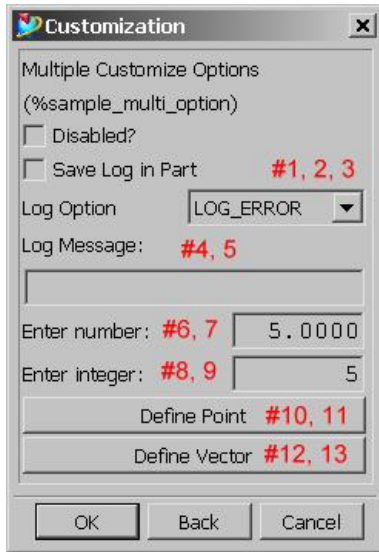
- `_label` = a string to define the label for the user interface element
- `_option` = a list of strings to define choices for an option menu
- `_init` = a list of strings to define choices for a list
- `_range` = a list of numbers for the values of the scale

The following example creates several widgets on the **Customize** dialog:

```
# Attributes for this class
(integer parameter) log_type: 1; #1
(list) log_type_option: { LOG_ERROR, LOG_WARNING, LOG_INFO }; #2
(string) log_type_label: "Log Option"; #3
(string parameter) log_msg: ""; #4
(string) log_msg_label: "Log Message: "; #5
(number parameter) number_input: 5; #6
(string) number_input_label: "Enter number: "; #7
(integer parameter) integer_input: 5; #8
(string) integer_input_label: "Enter integer: "; #9
(point parameter) point_input: point(0,0,0); #10
(string) point_input_label: "Define Point"; #11
(point parameter) vector_input: point(0,0,0); #12
(string) vector_input_label: "Define Vector"; #13
```

1. The `log_type`: parameter, along with the next (`log_type_option`: #2), create a dropdown menu for selecting items. The `log_type`: parameter stores the index of the item selected from the dropdown. The default item in the list will be the first item because this parameter is set to 1 in the dfa code.
2. The `log_type_option`: attribute is linked to the previous (`log_type`:) because it has the same name plus the `_option` suffix. This parameter contains the list of available items to be selected from the dropdown menu. The parameter qualifier is not used here because it is already used by the parameter that this one is linked to.
3. The `log_type_label`: attribute is also linked to the `log_type`: parameter by the suffix `_label`. This defines the label that will appear next to the dropdown menu.
4. The `log_msg`: parameter defines a text entry box. This parameter is typically used to allow the user to specify additional information to be printed in the log for this checker. The `do_check`: attribute must add this attribute to the string passed to the `ug_mqc_log` function for this to happen.
5. The `log_msg_label`: attribute defines the label for the previous parameter.
6. The `number_input`: parameter creates a numeric key-in field.
7. The `number_input_label`: attribute defines the label for the previous parameter.
8. The `integer_input`: parameter creates an integer key-in field.
9. The `integer_input_label`: attribute defines the label for the previous parameter.
10. The `point_input`: parameter creates a button which will allow users to define a point through the standard point constructor.
11. The `point_input_label`: attribute defines the label of the previous button.

12. The `vector_input`: parameter creates a button which will allow users to define a vector through the standard vector constructor.
13. The `vector_input_label`: attribute defines the label of the previous button.



Logging Results

The `ug_mqc_log` function is used to log the results of a check. The call to this function should appear at the end of the `do_check`: attribute definition block.

```
ug_mqc_log(nErrorLevel, lThreads:, $sUserMsg + $sLogString);
```

The first argument (`nErrorLevel`) should be `LOG_ERROR`, `LOG_WARNING`, `LOG_INFO` or `LOG_NONE` (name attributes defined in the `%ug_base_checker` mixin). This indicates whether the log will indicate an error, warning, info or pass. The second argument (`lThreads:`) should be a list of tags from one of the geometry query functions or an empty list `{}` if no geometry queries are being used (see tags section below). The last argument should be the string indicating the results of the check. If the check provides for a user defined message (`log_msg`: in the earlier example), this should be added to the message that is passed to `ug_mqc_log`. You can build very detailed messages to add to log file.

```
$check << if (!isEverythingOK?:) then @{
  $looks_like_non_dwg_name_with_drawing <<
    if (isDrawing?: & (passed_name_standard:="ppt or asm")) then
  ("This part meets the piece part or assembly~n" +
   "name spec, but also contains a drawing.~n")
    else ("");
  $looks_like_drawing_name_with_no_drawing <<
    if (!isDrawing?: & (passed_name_standard:="drawing")) then
  ("This part meets the drawing name spec, but~n" +
   "does not contain a drawing.~n")
    else ("");
  $looks_like_drawing_name_with_no_components <<
    if (isPiecePart?: & (passed_name_standard:="drawing")) then
  ("This part meets the drawing name spec, but~nhas no components.~n")
    else ("");

  $detail_msg << $looks_like_non_dwg_name_with_drawing +
    $looks_like_drawing_name_with_no_drawing +
    $looks_like_drawing_name_with_no_components +
    name_length_failure_str: + name_standard_failure_str:;
  ug_mqc_log( LOG_ERROR, {}, $usr_msg + $detail_msg + $debug_msg );
};
```

Logging is integrated into all `ug_mqc_check*` functions e.g.:

```
ug_mqc_checkCategoryLayer(String $category, String $layers, Name $log_type,  
String $log_msg)  
ug_mqc_checkRequiredRefSets(list $required_reference_sets, name $log_type,  
string $log_msg)  
ug_mqc_checkLayerEntity(String $layers, List $valid_entity_type, Name $log_type,  
String $log_msg )
```

Interactive log files

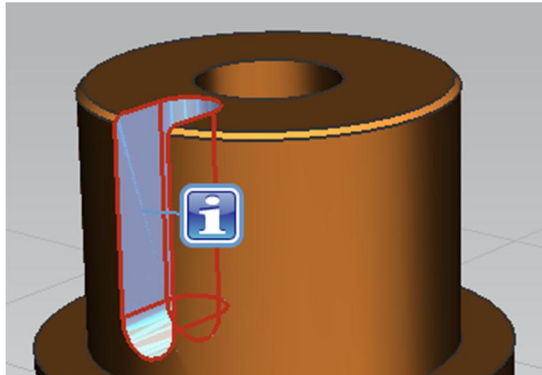
- One file containing results for all parts in the run
- Filename is timestamped for trend information
- Subsequent runs create new log with new timestamp

Batch log files

- One log file per part file
- Filenames are derived from part name
- Filenames are NOT timestamped
- Files are overwritten on subsequent runs

Logging Checker Tags

Many Check-Mate functions return tags (integers) to list items that have been selected by the function. These tags should be passed to the log function at the end of the `do_check:` attribute. This allows the user to visually see geometry that fails a checker by selecting the entry in the Results.



Use `mqc_XXX` functions to cycle and filter objects...

```
(any uncached) do_check: @{  
  # get a list of thread features in this part  
  $lThreads << mqc_askFeaturesByType("THREADS");  
  
  $sLogString << "Part contains detailed thread feature(s)";  
  
  # if the list of detail threads is not empty, then log an error  
  # pass the list of detail threads to the log so thread features can be easily  
  # seen  
  if (!empty?(lThreads:)) then @{  
    ug_mqc_log(LOG_ERROR, lThreads:, $sUserMsg + $sLogString);  
  } else @{  
    donothing;  
  };  
};
```

Special Attributes for Checks

The `%do_doc:` attribute can be used to provide extended documentation for a checker. You can specify a text string, a URL or a filename to display when using RMB on the result of a check.

```
(List) %do_doc: {name, string};  
# {URL, "http://anywhere.anything/pageX"}  
# {TEXT, "text to show"}  
# {FILE, "d:\textFile.txt"}
```

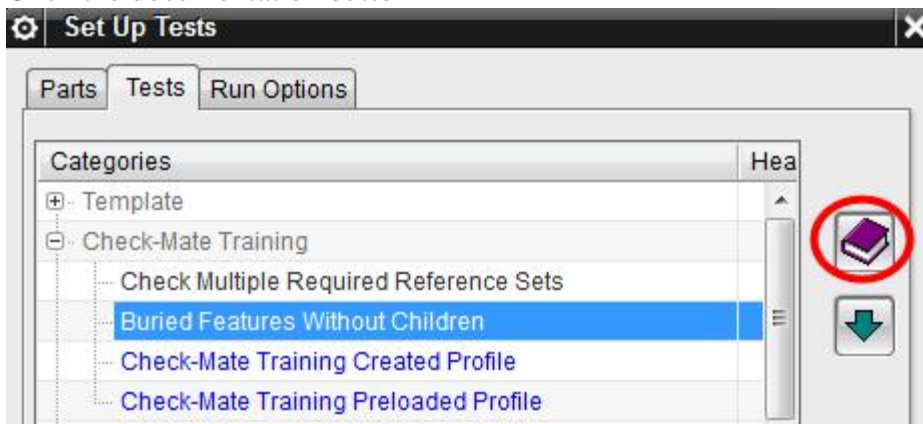
You can prevent or allow Check-Mate authors from making any modifications to a Check

```
(Boolean) allow_author_changes: true;  
# Set it to false to prevent modifications through the author tools.
```

Activity 14 - Understanding the Check-Mate dfa

In this activity we will examine a checker dfa file in detail to understand how the code affects the operation of the check.

1. Open the file `cm_train_check_buried_feature_without_children.dfa` in a text editor (Textpad is preferable to Wordpad, Word or Notepad) or ICE.
2. Look at the first line of code. What does it tell us?
`#! UGNX/KF 2.0`
This dfa file is compatible with NX version 2
3. What does the 2nd line do?
`DefClass:`
`%cm_train_check_buried_feature_without_children(%ug_base_checker);`
This defines the classname for the checker, remember the % prevents this class from being visible in the KF Navigator, Add Child Rule dialog.
4. What is the text between `#+` and `#-` for?
 - a. In NX, open the Check-Mate dialog
 - b. Select the Tests tab.
 - c. Select the Buried Features Without Children test (from Check-Mate Training)
 - d. Click the documentation button

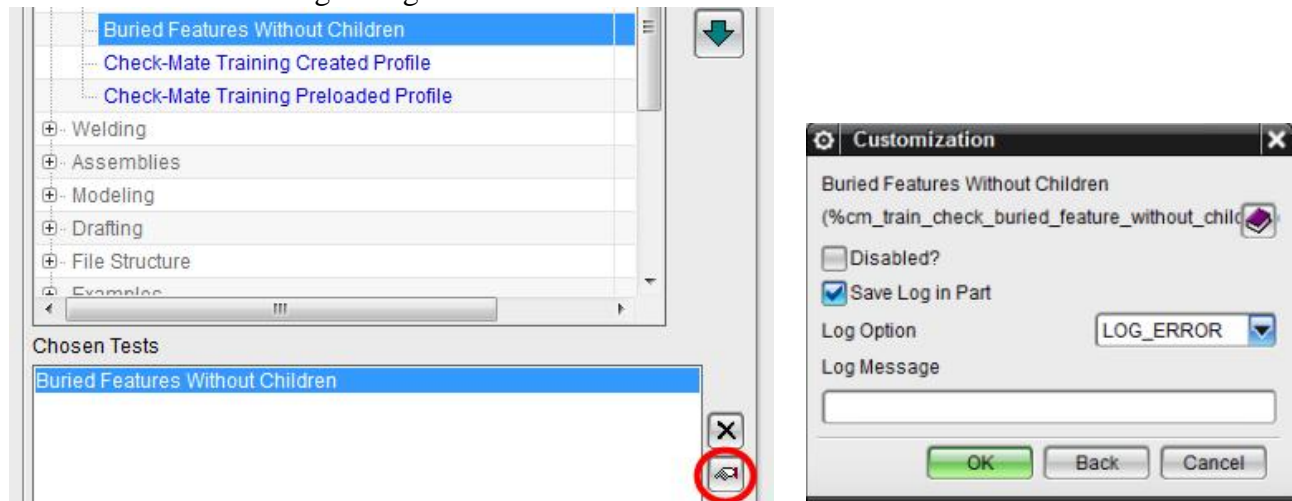


Note how the contents of the Info window corresponds to the comments

5. The next two attributes define how the test appears in the Category listing.
`(string) %test_category: "Check-Mate Training";`
`(string) %displayed_name: "Buried Features Without Children";`
What happens if you change the word “Check-Mate Training” to “Check-Mate Training.Part Files” in the `%test_category` attribute?
 - a. Make the change
 - b. Save the dfa file
 - c. Perform Reload All in the KF Navigator
 - d. Run Check-Mate and look at the Check-Mate Training category. You should see a new “Part Files” sub-category.
6. The next 7 attributes define some options for customizing the check when the user is running it. Remember that the parameter behavioral flag of the attribute data type specifier indicates that the attribute is allowed to be modified by an external source (such as a parent class or Check-Mate). Also recall how the `_label` and `_option` suffixes create a relationship between attributes.
`(boolean parameter) Disabled?: false;`
`(boolean parameter) save_log_in_part: true;`

```
# Attributes for this class
(integer parameter) log_type: 1;
(list) log_type_option: { LOG_ERROR, LOG_WARNING, LOG_INFO };
(string) log_type_label: "Log Option";
(string parameter) log_msg: "";
(string) log_msg_label: "Log Message";
```

From the Check-Mate Tests tab copy **Buried Features Without Children** to the Chosen Tests (by double clicking or selecting the test and clicking the green down arrow. Click the customize button and examine the resulting dialog.



Note how the attributes relate to the Customization dialog.

- a. **Disabled?** and **Save Log in Part** are boolean attributes and define toggle buttons on the dialog. Note that these do not have associated `_label` attributes. The label on the dialog is derived directly from the attribute name.
 - b. The 3 attributes beginning with `log_type` define a single item on the dialog; `log_type_label` defines the label, `log_type_option` generates the contents of the dropdown menu and `log_type` stores the line number of the item selected from the dropdown.
 - c. The `log_msg` attribute is a string type and defines a string keyin field.
 - d. What happens if you add a new parameter attribute?
 - e. What if you remove the `LOG_INFO` item from the `log_type_option` list?
 - f. What happens if you comment out all of the `log_type` attributes?
7. Examine the `do_check` attribute.
- a. There are several attributes which are collecting, sorting and filtering data (`all_feats_tags`, `all_buried_features`, `all_buried_with_bodies` and `all_buried_without_children`) These are used to determine what, if anything, fails the test.
 - b. The following statement checks to see if the list of failed items (from the attributes listed above) is empty. If it is not, log the failure, otherwise do nothing.

```
if (!empty?($all_buried_without_children)) then @{...
```
 - c. The 1st line after the if statement is building a detailed message to display in the validation results. It is a good idea to make this message as detailed as possible, including information regarding how to remedy the problem.

```
$detail_msg << "Found " + format("%d",
length($all_buried_without_children)...
```
 - d. The next line merely adds the detailed message to the log message (which can be added by the user).

```
$usr_msg << if (" = log_msg:) then "" else log_msg: + "~n";
```


- e. The next statement (which is separated into 3 lines to highlight the 3 arguments) is the one that logs the failure with Check-Mate. Pay particular attention to the arguments that are passed to the `ug_mqc_log` function. (Note that the following is all on one line in the `dfa` file)
- ```
ug_mqc_log(nth(log_type:, log_type_option:),
 $all_buried_without_children,
 $usr_msg + $detail_msg);
```
- i) The 1<sup>st</sup> argument defines the severity level. Look at this in more detail. The `nth` function returns an item from the `nth` position in a list. The 1<sup>st</sup> argument (`log_type`) is the position of the element to return. In this case `log_type` (discussed previously) stores the level of severity chosen by the user (or 1 by default). The second argument (`log_type_option`) is the list to select from. So the result here is that if `log_type = 1`, `LOG_ERROR` is passed to `ug_mqc_log`, if `log_type = 2` `LOG_WARNING` is passed and so on.
- ii) The 2<sup>nd</sup> argument is a list of failed items. It is very important to define this argument properly. This should always be a single level list containing only tags of geometric items (things that can be selected from the graphics window) that fail the test. This list of tags allows Check-Mate to locate the offending items (the user can easily navigate to and zoom in on the items from the validation results dialog). If the test is not evaluating geometry, you must pass an empty list `{ }` for this argument.
- iii) The 3<sup>rd</sup> argument is the error string which the user will see in the validation results when the test fails. This text can be quite long. Multiple lines can be constructed using `"~n"` as seen in the `$usr_msg` attribute.
- iv) Just after the `else` statement there is a single line containing the `donothing` statement. This is needed because every `if` statement must have an `else` statement. The `donothing` statement is used because if the test does not fail, we do not want to log any type of message in Check-Mate. In some cases, you may want to indicate that the test passed, in which case you could replace `donothing` with a statement similar to the one below:
- ```
ug_mqc_log(LOG_INFO, { }, "No buried features without children were  
found");
```

General Tips for creating Checks

Use the KF navigator when creating custom checks. Create test cases that intentionally cause the check to fail in interesting ways, and then add attributes to build the combination of function calls that will correctly filter down to the right set of failing objects.

Checks frequently start with one of the functions that collect "all" of a certain type of object, and then selectively either eliminate cases that are good (with the failures then remaining to display to the user) or selectively identifying the failures directly (again, collecting them to be displayed to the user.)

Most functions starting with either `mqc_ask` or `ug_mqc_ask` are good for collecting a set of objects as a starting point. Particularly useful for this are functions like `ug_mqc_ask_entities`, `mqc_askFeatures`, `mqc_askEntityOnDrawing`, `mqc_askObjectOfType`, etc.

Functions such as `mqc_selectEntitiesWithFilters` and `mqc_collectEntitiesWithFilterOptions` help either eliminate or collect objects of a certain type. Functions starting with `mqc_is` (e.g.: `mqc_isPiecePart` or `mqc_isLinkedFeature`) are good functions to use for Boolean "yes/no" to split a group of objects.

The majority of custom checks are a combination of multiple standard checks to arrive at a desired smaller subset. If this is true for a custom check you are working on, you can look at the original checks and extract the function calls used there to select the desired objects. Running these functions in series against an appropriate starting list often gets you exactly what you need.

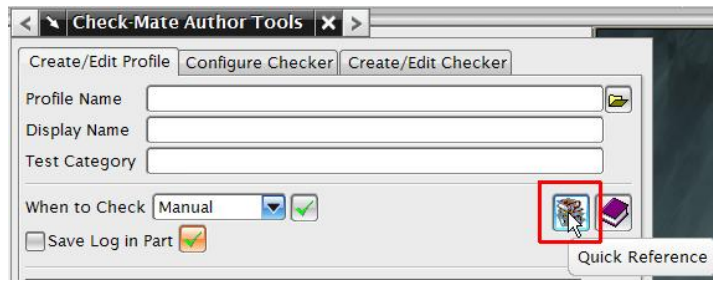
Whenever there are visible objects involved in the failure, try to populate the `object_tags` list in the `ug_mqc_log` function so that the individual entities will be identifiable in the results tab.

If you are directly identifying a case that is known to be bad for a particular reason, then try to incorporate an error string into the result that describes the exact condition that is causing the failure, just to make things easier to find and to fix.

Finding the Functions You Need

Where might you look to find Knowledge Fusion functions that can be used to author new Check-Mate checkers?

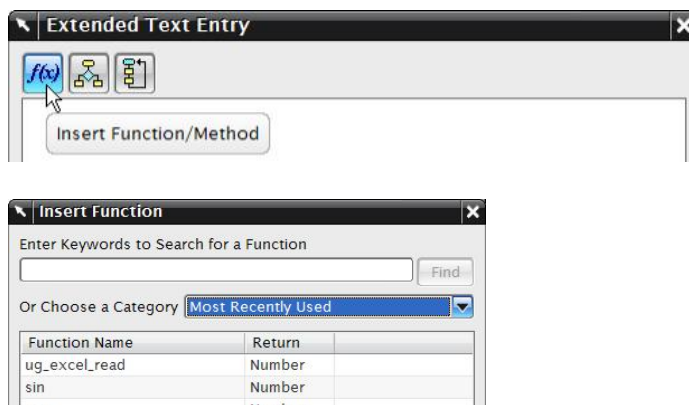
1. Check-Mate *Author Tests* dialog
 - a. Check-Mate Quick Reference



Generates (on the fly) a nicely formatted HTML reference guide listing all checkers and functions currently available in your session. (Note that this will also include any of your user-defined classes or functions that are active in your session.)

2. Knowledge Fusion *Add Attribute* dialog

The *Insert Function/Method* dialog in the *Extended Text Entry* area under *Add Attribute* in the Knowledge Fusion navigator is another rich source of Knowledge Fusion functions that can also be used in Check-Mate. This dialog also has a convenient search function built in.

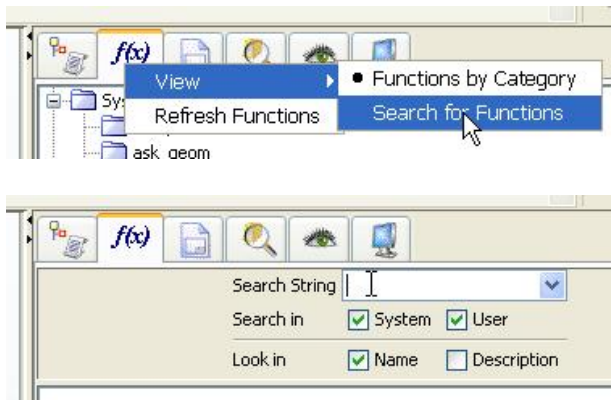


3. ICE Search

The *Function Explorer* inside ICE is another location that will dynamically show you the set of currently available functions.



An MB3 action on the **Function Explorer** tab will switch you to a search mode that looks at not just function names, but the full text of the function documentation as well.



4. Search Source Code

One very direct approach is directly searching the text of the DFA source code files located in the `/UGCHECKMATE/dfa/` folder in the NX install. This can be done using the regular search utility. Some programming editors also provide a specialized function for doing this.

5. Check-Mate and Knowledge Fusion Documentation

Specifically for programming style and technique questions, the docs are getting better and better. 😊

6. Yes, even NX Open

When all else fails, and you can't find a Knowledge Fusion function that does what you want, you can almost always fall back on the NX Open API. We'll talk more about this in a later section.

Customizing Check-Mate Feedback in HD3D

The presentation of HD3D Check-Mate results is now quite configurable, using a set of template files and a keyword query mechanism. In addition to the two “Info View” modes, Check-Mate tooltips and even the results summary visible on the HD3D palette are also configurable, as described below.

The following template files are used to configure each display:

	Tooltips
Console	hd_console_description_template.xml
Part	hd_part_description_template.xml
Test	hd_test_description_template.xml
Object Set	hd_objectset_description_template.xml
Object	hd_object_description_template.xml
	Info View – Less Detail
Part	hd_part_summary_template.html
Test	hd_test_summary_template.html
Object Set	hd_objectset_summary_template.html
Object	hd_object_summary_template.html
	Info View – More Detail
Part	hd_part_detail_template.html
Test	hd_test_detail_template.html
Object Set	hd_objectset_detail_template.html
Object	hd_object_detail_template.html

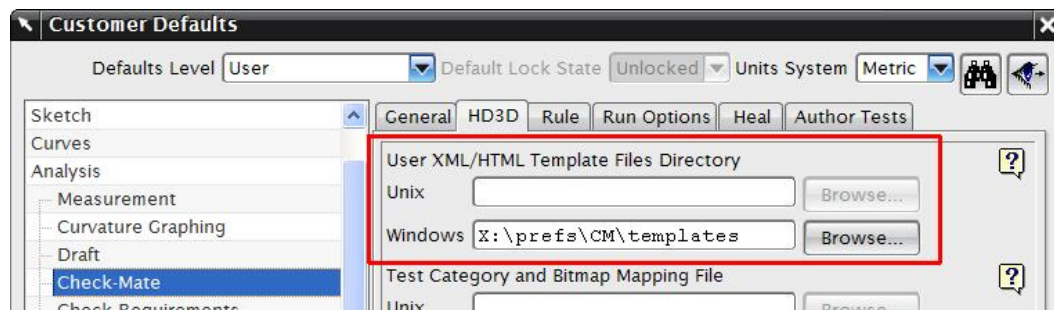
By default, NX finds these files in the **\DESIGN_TOOLS\checkmate\customization** folder in the NX install. If you choose to modify these templates, you should not edit the originals in the NX installation folder (as these will likely be overwritten by upgrade patches, for instance.) Rather, you should make a copy of the files you would like to modify and then make your modifications to the copy.

If you only want to tweak one or two of the templates, that’s fine – just copy the ones you want to adjust, and NX will find the rest of the templates back in the usual spot.

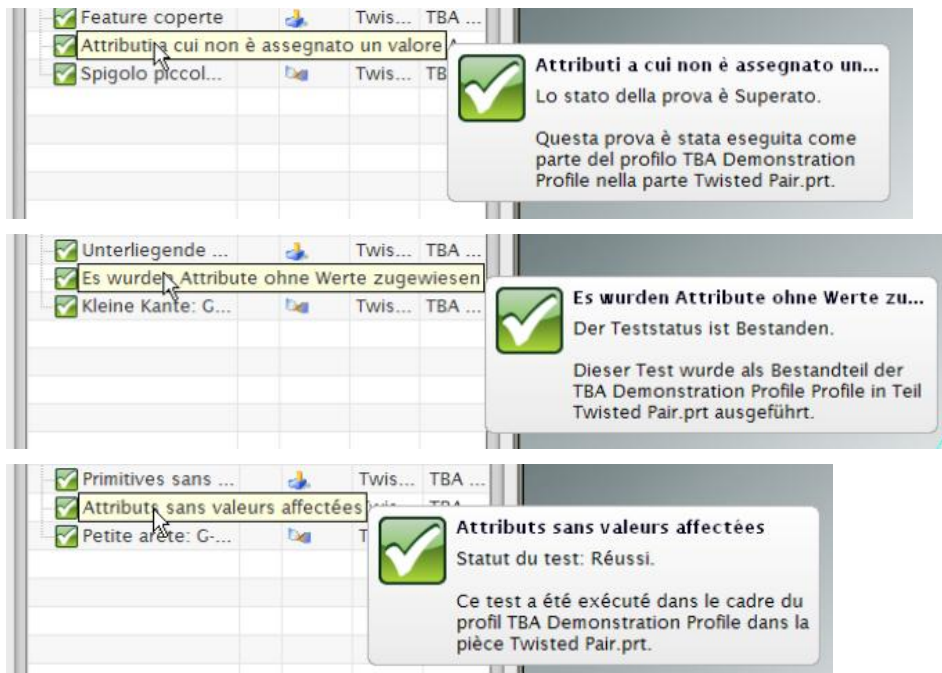
Of course, if you’re planning to deploy the modified templates across a body of users, you should place the modified templates in a common network location visible to all users.

In testing and deploying your modifications, you can use one of two methods for telling NX where to find your modified templates:

- The environment variable **UGCHECKMATE_TEMPLATES_DIR** can be used to locate the folder containing your templates, or
- You can use the “User XML/HTML Template Files Directory” customer default located in the **HD3D** tab under **Analysis** → **Check-Mate** to locate the folder containing your templates.



Out-of-the-box, these template files have been provided in eight different languages besides English. (...or nine, if you count both variants of Chinese in addition to French, German, Italian, Japanese, Korean, Russian and Spanish.) Templates for these languages can be found in subfolders within the **UGCHECKMATE_DIR/customization** folder. (Note that the folder for the correct language should automatically be selected if you are setting the environment variable **UGII_LANG**.)



For more detailed information on exactly how you can modify each of these templates, check out the “**readme.doc**” file that has been provided as part of the NX install in the **\DESIGN_TOOLS\checkmate\customization** folder.

Section 6: Checker Debugging and Troubleshooting

There are several places where errors can be detected while developing Check-Mate checkers.

- Special text editors with syntax highlighting capability.
- Reload All in the Knowledge Fusion Navigator
- The NX log File
- Debugging print statements.

Special Text Editors

Certain text editors can color code and automatically indent when you are creating/editing dfa files. The color coding and automatic indenting can help you find syntax errors in your code before you try to execute the checker.

The Interactive Class Editor (ICE) is a Knowledge Fusion Editor that is integrated into NX. It is similar to a standard text editor except that it has enhanced features for working with KF classes. The major functionality of ICE includes the following:

- Ability to edit and save DFA class files
- A context sensitive DFA file editor
- Typing completion for units, keywords, methods, and functions
- Model Viewer to visualize part geometry for standalone operation
- Drag and drop for adding class instances, KF functions, and template code to a DFA file
- Text adoption from geometry in NX to the DFA file
- Customize categories of classes or functions
- Customize an icon to represent the class or function
- Check DFA syntax and show the line containing the error
- Run either standalone or from interactive NX.
- Integrated KF Debugger in order to debug KF classes at run time
- Table Viewer for KF classes
- Evaluation of Temporary Rules

You can find the ICE editor:

- Click Class Editor on the Knowledge Fusion toolbar, or
- Choose Tools► Knowledge Fusion► Class Editor

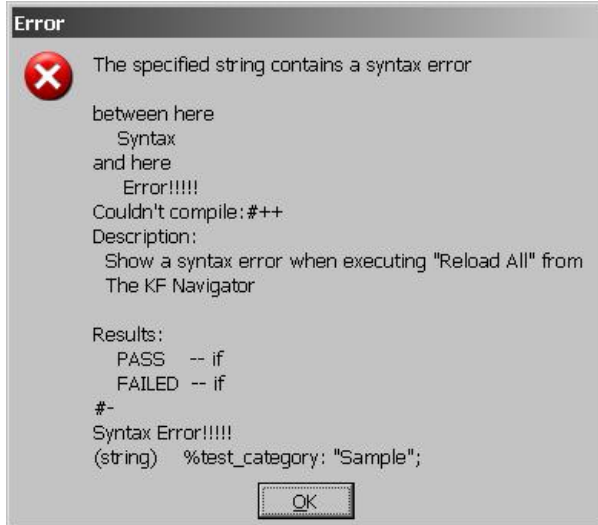
Knowledge Fusion Navigator

There are several tasks you can perform to debug a checker using the KF Navigator.

- Use **Reload All** to get Knowledge Fusion to read you latest changes
- Add individual attributes to the KF Navigator.
- Load your entire checker into the KF Navigator using **Add Child Rule...**

Reload All

While you are editing your checker dfa, you can use **Reload All** to get Knowledge Fusion to read your latest changes. Hint: never modify dfa code when NX is not running. When NX starts up, it reads all of the dfa files, but this process does not report syntax errors in a pop-up dialog. You may be able to find your errors in the NX log file, but they may be difficult to find and do not always appear there. You may not even realize there is an error until you look for your checker in the Check-Mate categories and you cannot find it. It is much better to start NX, modify your code, and then perform **Reload All**. This will provide immediate feedback in a pop-up dialog. For example:



Add Attribute

To test individual attributes you can add the attribute interactively in the KF Navigator as we did in Section **Error! Reference source not found.** activities. You can also test functions that are already in a dfa file by adding an attribute that uses the function.

Add Child Rule

If you have several attributes, it would be tedious and time consuming to add each one individually in the KF Navigator. It would be easier if you could add all of your code by using **Add Child Rule...** Unfortunately it is not normally possible to create an instance of a check class through the KF Navigator. The check classes don't appear in the class list because of the class name starts with %. Even if you remove the % from the name temporarily, Check-Mate classes still will not instantiate properly because of the mixin. You can bypass this for debugging by modifying the **DefClass** statement to follow standard Knowledge Fusion syntax. You must remove the % and change the mixin from `%ug_base_checker` to `ug_base_part`. Because of a little quirk in the KF parser; you cannot just comment the **DefClass** statement (the comment is ignored if the **DefClass** string is present on the line). In order to make it easy to switch back and forth between debugging and testing, you can use the following syntax:

For debugging in the KF Navigator:

```
#efClass: %cm_train_check_broken_wave_links(%ug_base_checker);  
DefClass: Report_Broken_Wave_Links(ug_base_part);
```

For testing in Check-Mate

```
DefClass: %cm_train_check_broken_wave_links(%ug_base_checker);  
#efClass: Report_Broken_Wave_Links(ug_base_part);
```


Note how, in both cases, the 'D' of DefClass is overwritten with the # so the line will be treated as a comment. When you have your dfa syntax as in the 1st example above, you will be able to add your Check-Mate class in the KF Navigator (RMB – Add Child Rule...). You can check the value of each attribute by using RMB – Show Value in the KF Navigator.

Once you have your checker instantiated, you can use Show Value on the attributes to determine what the results are.

NX Log File

If your class does not appear in the Categories list, check the NX log file (Help► NX Log File). Use the find functionality of the Information window to search for your class name (e.g. %test_class). You may find an error e.g.:

```
Cannot get class: '%test_class'
between here
    Here
and here
    is a syntax error!
Couldn't compile:#++
Description:
    Show a syntax error when executing "Reload All" from
    The KF Navigator

Results:
    PASS - if
    FAIL - if
#-
Here is a syntax error!
(string) %test_category: "Sample"
(string) %displayed_name: "Syntax Error";
```

Using Debugging Print Statements

You can also use debugging print statements to determine what the values of attributes are during checker execution. Remember that you can use standard KF functions in your Check-Mate classes. The `ug_printValue` function is the simplest function to use for debugging. This function will pop up the Information window to print the values, making it easy to check. Note that this function will print any type of data. You can build detailed output messages by adding bits and pieces of data together. You must be careful though, you cannot add dissimilar data types. Typically you would build a string by using the `format` or `stringValue` functions to convert other data types into strings as shown in the following examples:

```
# list of failed items
$failed_items << $failed1 + $failed2;
# print the number of items
ug_printValue("number failed: " + length($failed_items));
# a different way to print it
ug_printValue(format("number failed: %d", length($failed_items)));
# print each failed tag
Loop {
    for $i in $failed_items;
    # because of loop syntax, must use do statement
    do ug_printvalue($i);
};
```

Code Organization

There are two schools of thought when deciding how to code Check-Mate classes: perform calculations inside the `do_check` or outside of the `do_check` attribute.

Performing Calculations Inside `do_check`

When using this coding practice, you would perform all of the calculations for the check in attributes inside the `do_check`,. For example:

```
# Checker function
(any uncached) do_check: @{
  $all_feats_tags << mqc_askFeatures();
  $all_feature_bodies << loop {
    for $tag in $all_feats_tags;
    collect mqc_askBodyOfFeature($tag);
  };

  $all_buried_features << loop {
    for $tag in $all_feats_tags;
    collect mqc_isFeatureBuried($tag);
  };

  $all_buried_with_bodies << loop {
    for $tag from 1 to length($all_feats_tags);
    if ((nth($tag,$all_feature_bodies) != 0) & (nth($tag,$all_buried_features)))
      collect nth($tag,$all_feats_tags);
  };

  $all_buried_without_children << loop {
    for $tag in $all_buried_with_bodies;
    if (length(second(mqc_askFeatureRelatives($tag))) = 0) collect $tag;
  };

  if (!empty?($all_buried_without_children)) then @{
    $detail_msg << "Found " + format("%d", length($all_buried_without_children))
+
    " buried feature(s) without children";
    $usr_msg << if (" = log_msg:) then " else log_msg: + "~n";
    ug_mqc_log(nth(log_type:, log_type_option:),
      $all_buried_without_children,
      $usr_msg + $detail_msg);
  } else @{
    donothing;
  };
};
```

Here you see several attributes defined and used inside the `do_check` attribute (`$all_feats_tags`, `$all_feature_bodies`, `$all_buried_features`, `$all_buried_with_bodies`, `$all_buried_without_children`).

The advantages to this approach are:

- Most of the standard Check-Mate code uses this practice, so when you copy parts of these classes, you will get this type of format.
- Debugging with `ug_printValue` is somewhat simplified vs. performing calculations outside `do_check`. This is described in more detail in the next section.

The disadvantages of this approach are:

- When debugging, you will need to use `ug_printValue` to evaluate the results of the attributes in the `do_check`.

- When trying to evaluate list contents, you will have to create a loop to print each value of the list individually. This is particularly problematic when there are multi-level lists.

Performing Calculations Outside do_check

When using this coding practice, you would perform all of the calculations in attributes outside of the do_check, then reference these inside the do_check. For example:

```
(list) all_feats_tags: mqc_askFeatures();
(list) all_feature_bodies: loop {
  for $tag in all_feats_tags;;
  collect mqc_askBodyOfFeature($tag);
};

(list) all_buried_features: loop {
  for $tag in all_feats_tags;;
  collect mqc_isFeatureBuried($tag);
};

(list) all_buried_with_bodies: loop {
  for $tag from 1 to length(all_feats_tags);
  if ((nth($tag,all_feature_bodies:) != 0) & (nth($tag,all_buried_features:)))
  collect nth($tag,all_feats_tags);
};

(list) all_buried_without_children: loop {
  for $tag in all_buried_with_bodies;;
  if (length(second(mqc_askFeatureRelatives($tag))) != 0) collect $tag;
};

# Checker function
(any uncached) do_check: @{
  if (!empty?(all_buried_without_children:)) then @{
    $detail_msg << "Found " + format("%d", length(all_buried_without_children:))
+
    " buried feature(s) without children";
    $usr_msg << if (" = log_msg:) then "" else log_msg: + "~n";
    ug_mqc_log(nth(log_type:, log_type_option:),
               all_buried_without_children:,
               $usr_msg + $detail_msg);
  } else @{
    donothing;
  };
};
```

Here you see several attributes defined outside and used inside the do_check attribute (all_feats_tags:, all_feature_bodies:, all_buried_features:, all_buried_with_bodies:, all_buried_without_children:).

The advantages to this approach are:

- When debugging, you can add your checker as a child rule and use **Show Value** to evaluate the results instead of using ug_printValue. This makes it easier to check the values because you do not have to try to determine which line of the ug_printValue statements correspond to the attribute
- When evaluating list contents, all you will have to do is click the + box next to the list name to see everything in the list. This is particularly advantageous vs. using ug_printValue when it is a multi-level list.

The disadvantages of this approach are:

- Most of the standard Check-Mate code uses the other approach, so when you copy parts of these classes, you will not get this type of format.
- You will need to modify the DefClass (as discussed earlier) to instantiate the checker class.

- If you need to use `ug_printValue` to print intermediate results it may be somewhat more complicated in some cases because of the way KF evaluates blocks of code. Remember that the value assigned to an attribute is the value of the last statement in the block. This means that you probably will not want to have the `ug_printValue` as the last statement in the block because it will always return 0, which may be the wrong data type.

In the following activities we will use several techniques to investigate how Check-Mate code is executing. For each activity, make sure NX is already running; use the Textpad or ICE editor to make changes.

Activity 15 - Check for syntax errors

1. Edit `%cm_train_check_features_for_non_parametric_parents.dfa`
2. Look at the values for `%test_category` and `%displayed_name`.
3. In NX, start Check-Mate and see if you can find this checker in the **Categories** list. Why is it not there?
4. Open the NX log file (Help ► NX Log File). Search for `non_parametric`. What do you see?
5. Can you point out the syntax errors?

Activity 16 - Adding debug print statements

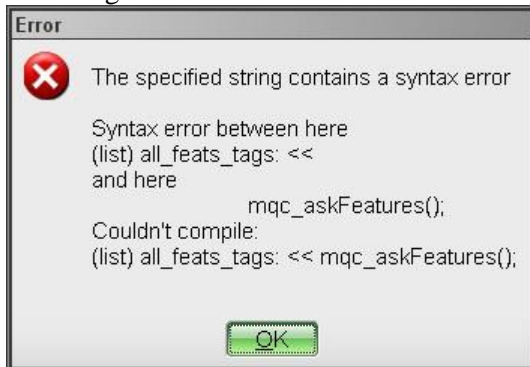
1. Edit `%cm_train_check_buried_feature_without_children.dfa`
2. Add a debug print message after the `$all_buried_features` assignment (after the closing `}`).
`ug_printValue($all_buried_features);`
3. Reload and run this test in Check-Mate. What does this output tell us?
4. Change the line you just added to use the `printList` function (supplied in **`cm_train_functions.dfa`**) instead of `ug_printValue`. Check the dfa for a description of the arguments to this function. Enter the following line exactly as shown :
`listPrint($all_buried_features, 0, " ");`
5. Reload and run this test in Check-Mate. What happens? Why is there no output from this function? Look at the results in the **Validation Results** dialog. What do the **Result Details** tell us?
ERROR: The specified string contains a syntax error
Error encountered during checker execution. Check syslog file for details.
6. Oops, we have introduced a syntax error. Unfortunately, Check-Mate does not always tell us what the exact problem is. If you look at **`cm_train_functions.dfa`** closely, you will see that the function is called `printList`, not `listPrint`. Fix the problem and test it in NX.
7. Now we can see what `$all_buried_features` is collecting.

Activity 17 - Add a check as a child rule.

1. Open %cm_train_check_buried_feature_wo_children.dfa
2. Edit the DefClass statement so that you swap the # and D of the 2nd two lines
#efClass: %cm_train_check_buried_feature_wo_children(%ug_base_checker);
DefClass: cm_train_check_buried_feature_wo_children(ug_base_part);
3. In NX use Add Child Rule to add your check in the KF Navigator. After you type in a name and select the check, why are the OK and Apply buttons grayed?
4. Open the NX log file. Search for wo_children. Oops, there is a syntax error
Cannot get class: 'cm_train_check_buried_feature_wo_children'
Syntax error between here
(list) all_feats_tags: <<
and here

mqc_askFeatures();

Couldn't compile:
(list) all_feats_tags: << mqc_askFeatures();
5. Fix this problem, reload and add the check in the KF Navigator.
6. Open the Attributes list of the class.
7. Click the + next to some of the all_... list attributes. See how this might be simpler than adding a bunch of ug_printValue, ug_printMessage or listPrint statements?
8. Note that the reason there was no 'in your face' warning about the syntax error is because the error existed before NX was started. Remember that errors found by NX when loading during startup are only written to the log file. When you are editing your dfa files it is a good idea to have NX already running to catch any new mistakes. Change the file back to the way it was originally and reload. You should see a dialog similar to the following



Section 7: Tips for Successfully Implementing Check-Mate

Overview

Proper deployment will require some planning and preparation on the part of system administrators and/or engineering managers. Proper deployment can significantly streamline the user experience and solidify standardization in model quality checking.

Remember, the main purpose of Check-Mate is to help enforce corporate standards, good modeling practices, verifying correct usage of reference sets, WAVE links, etc. As installed by default during the standard NX installation, Check-Mate is preconfigured to immediately start providing benefit to users. Over 300 checks are immediately available and ready-to-use without any additional configuration effort. In many cases, however, companies will want users to consistently be validating their designs using a predetermined set of company-specific validations. NX provides a variety of tools for making this process easier for the users. NX also provides an optional set of tools for making it harder for users to AVOID validation of their designs. Companies can choose the specific level of enforcement they would like to employ with respect to Check-Mate validation. The following sections will explain various ways to simplify Check-Mate configuration and make it easy for the users to access.

Environment and File Locations

If your company decides to pre-configure the Check-Mate environment to deliver company-specific checks and/or profiles to NX users, a very early consideration must be the location in which customized Check-Mate content will be stored. As you may recall NX has the ability to find Check-Mate content in a variety of places, depending on a company's deployment preferences. NX will actually automatically look in several places for checks – these various locations are detailed in **Error! Reference source not found.** (page **Error! Bookmark not defined.**). There are even some settings in the Customer Defaults (File ► Utilities ► Customer Defaults...) for Knowledge Fusion and Check-Mate. It is highly recommended that you DO NOT use Customer Defaults to define the search paths for Check-Mate. With this method it is not as easy to set up globally or to propagate changes as with environment variables.

The simplest way to set up Check-Mate environment variables is to define `UGCHECKMATE_CONFIG` to point to a local copy of `ugcheckmate.dat` from the NX installation in `ugii_env.dat` (for native NX) or in a startup batch file that everyone uses to start NX (or Teamcenter). The `ugcheckmate.dat` file should be located in a shared directory that all NX users can access. Once this is set up, any changes that need to be made can be done in this one file and all users will see the changes the next time NX is started.

You will probably want all NX users to have access to the same Check-Mate customizations. For this purpose, it is good practice to place all Check-Mate customization files under a single main

directory (with subdirectories below) on a shared drive that all Check-Mate users have access to (e.g. N:\NX_customization\Check-Mate\). Here are some suggested subdirectories and the reason why you may want to use them. **Important:** except where noted, you should protect these directories from modification by the NX users.

Subdirectory	Purpose
application	<p>Check-Mate will automatically look for certain files here.</p> <ul style="list-style-type: none"> • If you have any extended Check-Mate functions that use NX/Open (c, c++, java) code, you can place the dll files here and Check-Mate will find them. • This may also be a good place for the startup and other custom batch scripts.
config	<p>In a large installation, you may have multiple types of Check-Mate users who require different configurations. Here you could locate different versions of the <code>ugcheckmate.dat</code> file for the various users (perhaps with different definitions for variables like: <code>UGII_CHECKMATE_DEFAULT_CHECKER</code>, <code>UGII_CHECKMATE_ALLOW_AUTO_RUN</code>, <code>UGII_CHECKMATE_LOG_DIR</code>, <code>UGCHECKMATE_ALLOW_CHECKER</code>, <code>UGCHECKMATE_HIDE_CHECKER</code>, <code>UGCHECKMATE_ALLOW_CATEGORY</code> and <code>UGCHECKMATE_HIDE_CATEGORY</code>).</p>
data	<p>Certain checks may need access to external data (e.g.: spreadsheets or ODB) for comparison rules. Database, Excel or other data files could be located here.</p>
dfa	<p>Depending on which environment variables you have defined, you may or may not need a dfa directory. The dfa directory will be needed if you use the following environment variables to define the location to look for checker classes. <code>UGII_USER_DIR</code>, <code>UGII_SITE_DIR</code> or <code>UGII_VENDOR_DIR</code>. If you use <code>UGCHECKMATE_USER_DIR</code>, you do not need the dfa directory (however, it is good practice to separate the dfa file from other files if you have more than a few custom checkers).</p>
docs	<p>This is a good place for extended documentation for checkers. If you use the <code>%do_doc</code> attribute, you can point to files in this location. If you use html for extended documentation, you may wish to name this directory “html” instead. If you want your extended documentation to point to an intranet location, this directory is probably not needed.</p>
logs	<p>Use this directory to store checker log files. Define <code>UGII_CHECKMATE_LOG_DIR</code> to point to this location. If you want non NX users to be able to view the checker results using the external viewers, you may need to place this directory on another shared drive depending on which shared drives are available to these other users. In any case, this directory will need to have write permission for the Check-Mate users (or at least the NX process).</p>

Simplifying Check-Mate for Improved Efficiency

Check-Mate is a powerful tool for ensuring parts, assemblies and drawings conform to corporate standards. However, unless it has been configured properly, it can be quite intimidating for the general user. There are over 300 checks in the standard Check-Mate installation. It would certainly not be an effective use of time for the user to have to search through all of the checks

each time a part needs to be checked. This is why there are tools provided in Check-Mate to simplify user interactions. This customization is performed using a combination of environment variables and profiles.

Simple Organization

In order to improve the efficiency of Check-Mate users, it is always a good idea to create profiles to collect commonly used checks. Frequently checks are organized into groups based on tasks (part modeling, assembly, drafting) or user roles (engineer, analyst, draftsperson, etc.). When profiles are carefully designed, they can make Check-Mate use much easier. If you have savvy users or users who need to run other checks (not in profiles) occasionally, you can leave your Check-Mate configuration at this level.

Using Variables to Simplify the Interface

If your users do not need to run any checks other than those in the corporate profiles, you can remove all of the other Check-Mate categories using the environment variable `UGCHECKMATE_ALLOW_CATEGORY` set to only the category(ies) containing the desired profiles and checks. For example:

```
UGCHECKMATE_ALLOW_CATEGORY=Check-Mate Training
```

When the list of categories is displayed in Check-Mate, only the “Check-Mate Training” category will be visible. Multiple categories can be used by listing them separated by commas (,).

You can further simplify user interaction by causing the most commonly used check(s) and/or profile(s) to be pre-selected when Check-Mate is started. By setting the variable `UGII_CHECKMATE_DEFAULT_CHECKER`, the users may not even have to select checks or profiles under normal Check-Mate use. When Check-Mate is started, these profiles and/or checkers will already be selected and the users could just press the **Execute Check-Mate** button. Multiple checks and profiles can be used by listing them separated by commas (,). Note: use the class name, not the displayed name in the list. For example:

```
UGII_CHECKMATE_DEFAULT_CHECKER=%cm_train_profile_save_time,cm_train_profile_drafting
```

To further improve user productivity, you can set variables separately for different types of users to pre-select checks / profiles and show only the categories that apply to his/her role.

Complete Automation

You may want to set up a “smart” profile to use as the pre-selected item. In order to be “smart” the profile should be able to determine which type of file is being checked and only run the appropriate checks for that type. For example, the profile should not run modeling specific checks if the file is a drawing. Use the following code constructs to make the profile “smart”:

```
# Define a function or method to determine which type of file is being
checked
# use functions like mqc_isDrawingPart, mqc_isAssemblyPart, etc.
defun: checkType() @{ ... }integer;
# Define attributes to indicate which type of file it is
```

```

(boolean) fPrt: (checkType() = 1); # file is a part
(boolean) fDwg: (checkType() = 2); # file is a drawing
(boolean) fAsm: (checkType() = 3); # file is an assembly
(boolean) fIgn: (checkType() = 0); # ignore this file
# For each check in the profile, use the type check to see if the check
should run.
(child) %cm_train_check_retained_dimensions {
  class, if (fDwg) then %cm_train_check_retained_dimensions else
nulldesign;
};
# Note that in some cases a check should be run on more than one type of
file
(child) %cm_train_check_empty_part_attribute {
  class, if (fDwg | fPrt | fAsm) then % cm_train_check_empty_part_attribute
else nulldesign;
}; # you could also use if (!fIgn) above

```

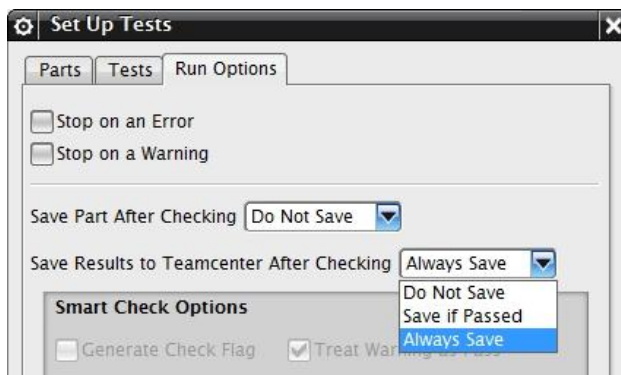
This type of customization is a bit complicated, but the user's life is greatly simplified.

Section 8: Check-Mate and Teamcenter

Storing Check-Mate Results to Teamcenter

When running NX 7 or higher in managed mode with Teamcenter 8 or higher, storing Check-Mate results into Teamcenter should be almost completely automatic.

Inside Check-Mate (or starting with NX 7.0, in the Customer Defaults for Check-Mate), you'll need to tell NX to save your Check-Mate results to Teamcenter. For production deployments, this should generally be done by setting the desired customer defaults for all users:



Why save immediately after checking?

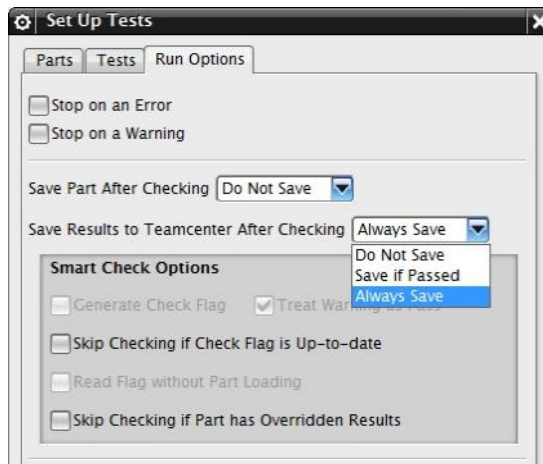
Depending on your preferences, Check-Mate may or may not automatically perform a save operation immediately after checking. Why would you want to do this? The biggest reason is that this is the only time when you can be sure that the results saved in the database are in sync with the part that is saved in the database.

If the part is not saved immediately after checking, then there will always be a possibility that the user:

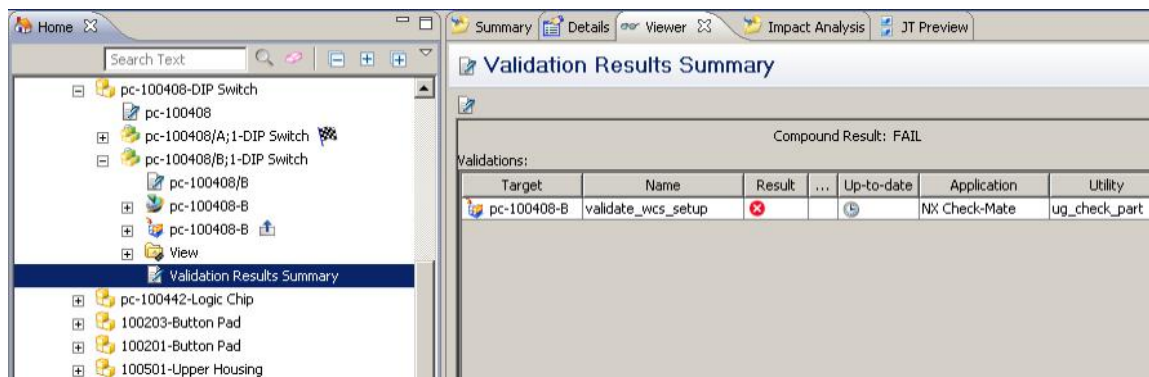
- 1) Checked the part,
- 2) Edited the part, and then
- 3) Saved the part

In this scenario, it should be obvious that the results **MIGHT** no longer be representative of what is actually in the part. To prevent this situation, we give you the option of eliminating the possibility of that intermediate editing by automatically performing the save right after the checking.

(Note that these same settings can also be changed interactively in the **Run Options** tab of the **Set Up Tests** dialog:



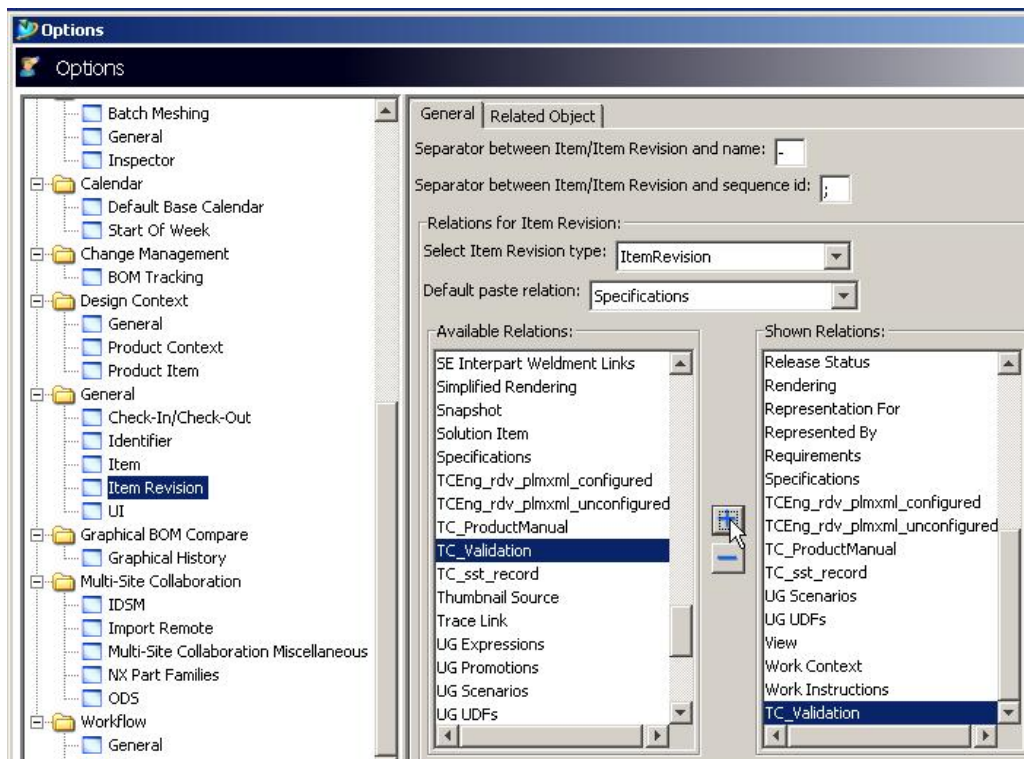
Once Check-Mate saves results to Teamcenter, these results should then be visible in the **Validation Results Summary** under each item revision:



What if I can't see the Validation Results Summary?

If the **Validation Results Summary** is not yet visible and you know that you have saved Check-Mate results to Teamcenter, then you may just need to “turn it on” by going to:

- a) **Edit** → **Options** in the Teamcenter pull-down menus, and then ...
- b) in the tree on the left side of the Options dialog, choosing:
Options → **General** → **Item Revision** and then...
- c) Using the (+) button to add the TC_Validation relation to the right column, as shown:



After you have done this the **Validation Results Summary** should be visible.

Section 9: Check-Mate External Viewers

Activity 18 – Check-Mate External Viewers

Step 1: Use Learning Advantage or Learn@Siemens :

- Check-Mate external viewers
 - Check-Mate log files
 - ✦ Create a Check-Mate log file
 - Quality Dashboard utility
 - ✦ Review results with the Check-Mate Viewer
 - ✦ Define a report template and display results

- Currently, Quality Dashboard only supports TC2007 and previous versions. So it is not possible to connect to Teamcenter 9.1, for eg, with Quality Dashboard.