

# Complexity Metrics Tool User Guide

---

Min Young Nam

## Introduction

This is a user guide for a set of complexity metric tools that have been built as TCL scripts to be installed on SCADE. SCADE supports these TCL scripts to be installed as extensions. This guide will explain how to run the tool on SCADE models and explain what part of the model the tool covers so that the user would understand what it does and recommend changes if necessary.

This guide will be updated as changes are made to existing tools and as additional tools are added. Some analysis tools are preliminary.

## Installation of TCL scripts in SCADE

TCL scripts are installed by registering the TCL script in SCADE which makes SCADE load the script when you restart the SCADE Suite. Changes to the TCL script will be applied automatically as long as the script file remains in the same file directory with the same file name. Registering TCL scripts will create a new menu named “Complexity Tools” and a menu item inside depending on script that is registered. In the following, we will give a detail step-by-step guide on how to install the TCL scripts. This is in part the same as following the instructions from Help Topics named “Registering Script Files from SCADE Suite IDE.”

[Editor](#) > [4 Customizing SCADE Suite Editor in TCL](#) > [Registering Customization Scripts](#) > [Registering Script Files from SCADE Suite IDE](#)

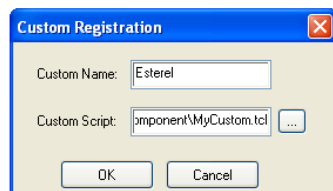
---

### Registering Script Files from SCADE Suite IDE

You can register your own scripts directly from SCADE Suite IDE.

#### To register custom script file

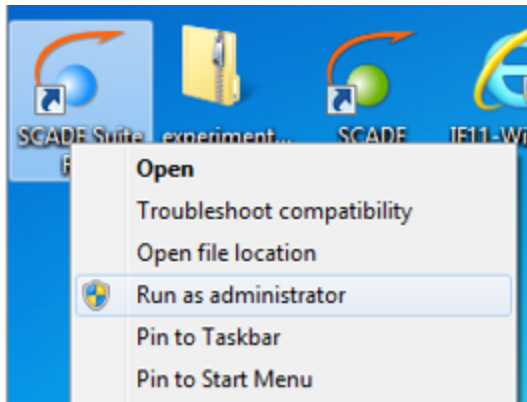
- 1 Select *Insert* > *Components* > *Custom* to open **Custom Registration**.



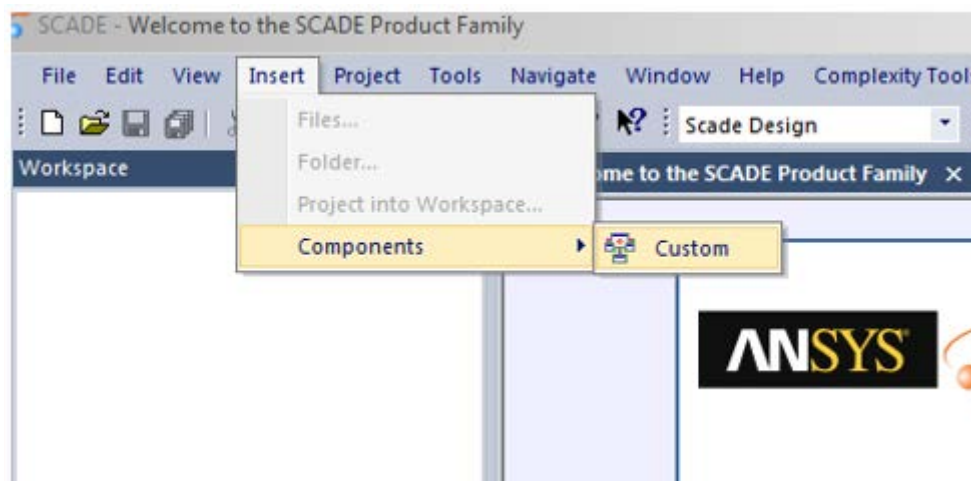
**Figure 4.1:** Custom Registration dialog

- 2 Type a name in the **Custom Name** field.
- 3 Set the path to the script file in the **Custom Script** field.
- 4 Click **OK**.
- 5 Close SCADE Suite and launch again. The custom script is automatically registered and evaluated at application start or upon script file update.

1. Save the directory of TCL scripts named “ComplexityTools” into any folder that you feel comfortable.
2. Start the SCADE Suite by right-clicking on the SCADE Suite icon and selecting “Run as administrator” for the context menu.

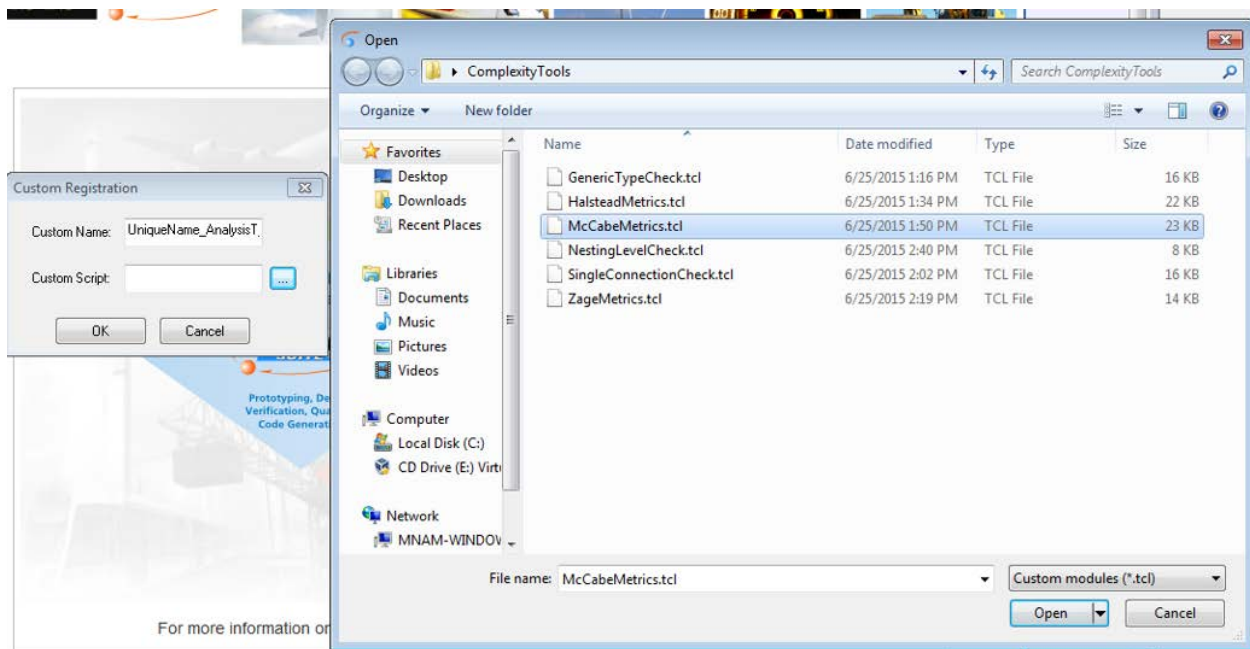


3. Once SCADE Suite finished loading, select “Insert > Components > Custom” from the main menu.

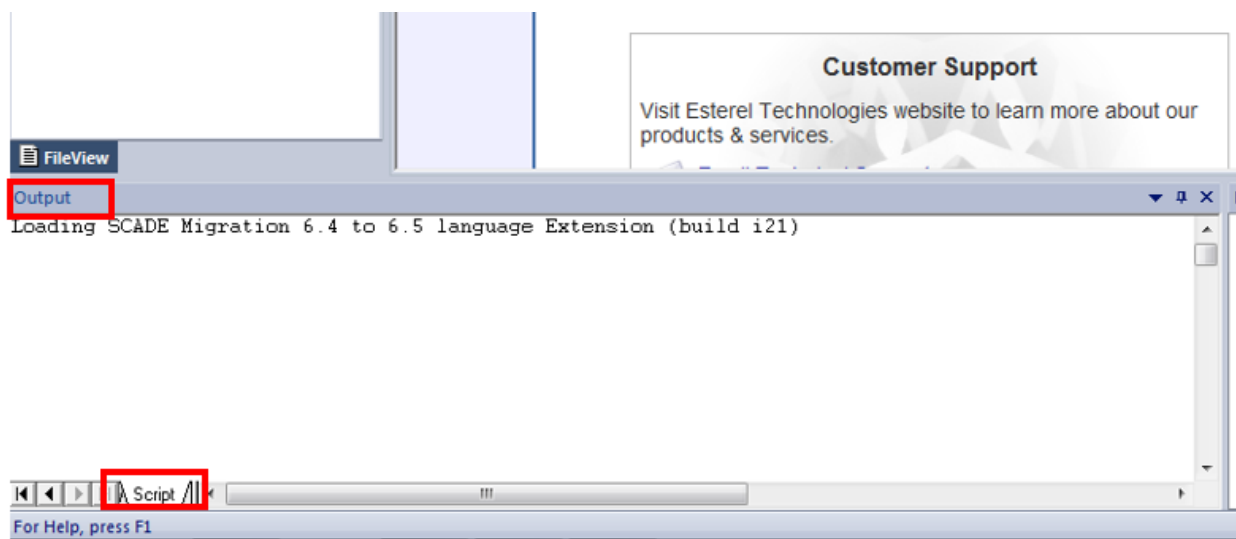


4. A small “Custom Registration” window opens where it asks for “Custom Name” and “Custom Script.” In the “Custom Name” box, write down a name for the TCL script that you are about to register. A name that is Unique and represents the type of analysis is recommended for easier uninstallation in the future. Ex) OrganizationName\_McCabeMetrics

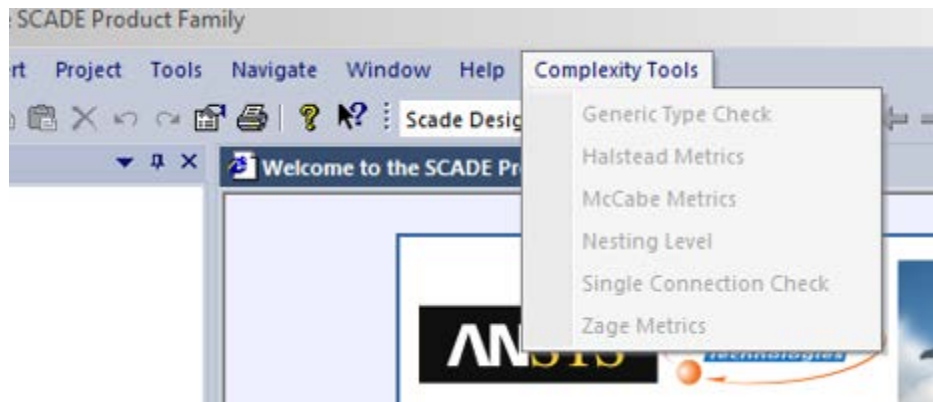
Then for the “Custom Script” box, click on the “...” box and select the TCL script file that you would like to install.



5. Click “OK” in the Custom Registration window.
6. Now close and restart SCADE. Doesn’t have to be run as administrator.
7. When SCADE restarts, it will load all registered TCL scripts. If the “Output” area shows no error messages in the “Script” tab, it means that all scripts have loaded correctly.

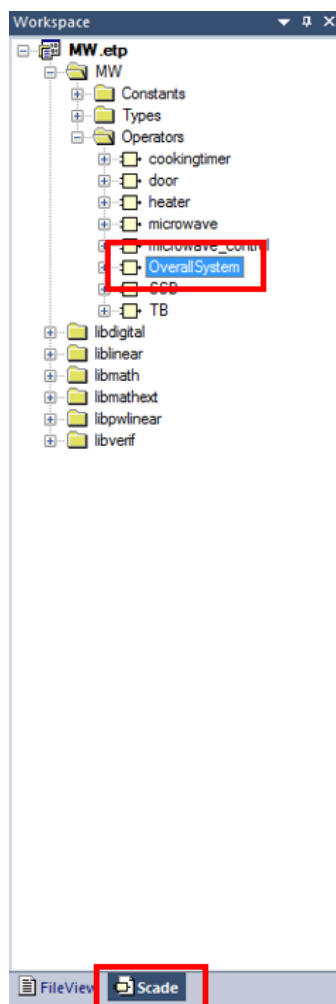


8. You will also recognize that a new menu item named “Complexity Tools” now exists. In the menu, any new script that is installed should be added to the list. After repeating these steps for every or a subset of the scripts that you would like to install, the menu would look as the following,

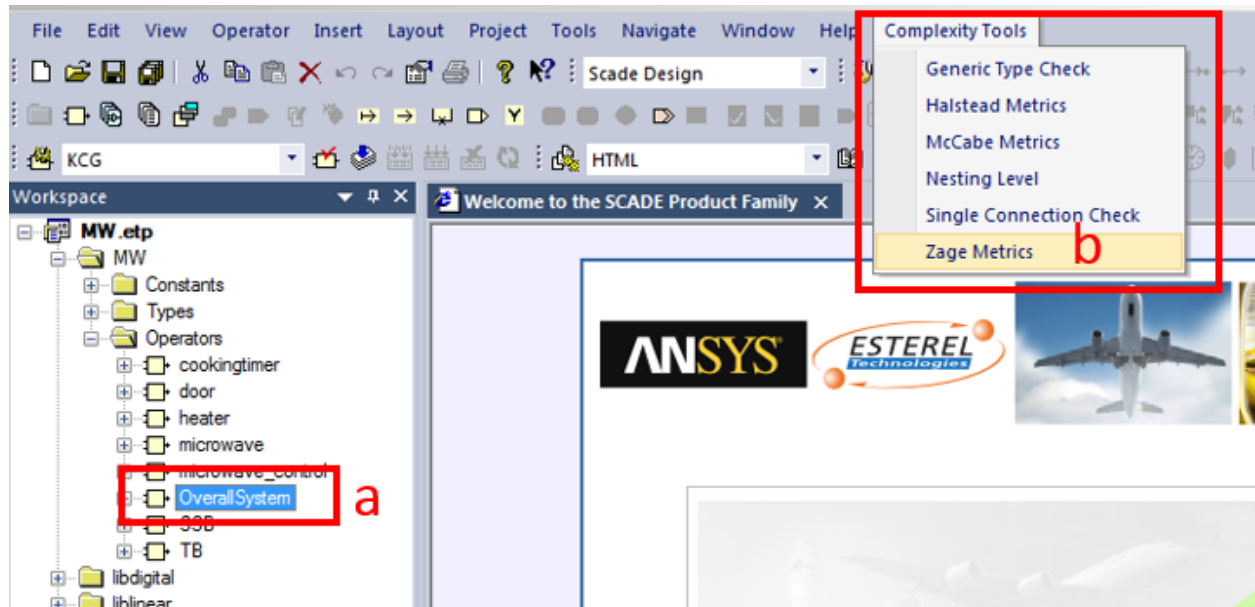


## Running Installed Complexity Tools

Running any of the Complexity Tools requires a sample workspace to be loaded. After loading a workspace, select any top level Operator that you would like to run the analysis for from the Workspace "Scade View."



Then immediate (without selecting anything else in the Workspace), select the tool that you would like to execute from the menu “Complexity Tools.” This will execute the script based on the selected file in the Workspace.



**Note that, unless otherwise mentioned, analysis is based on the selected operator. For example, if there is an operator that is defined but never used in the selected operator by any relation, it will not be part of the analysis.**

## Where Results are displayed

Generally, there are three types of places where the tool results are displaced.

1) Textual results are displayed in the “Script” tab which is located in the “Output” area which is generally located in the bottom left.



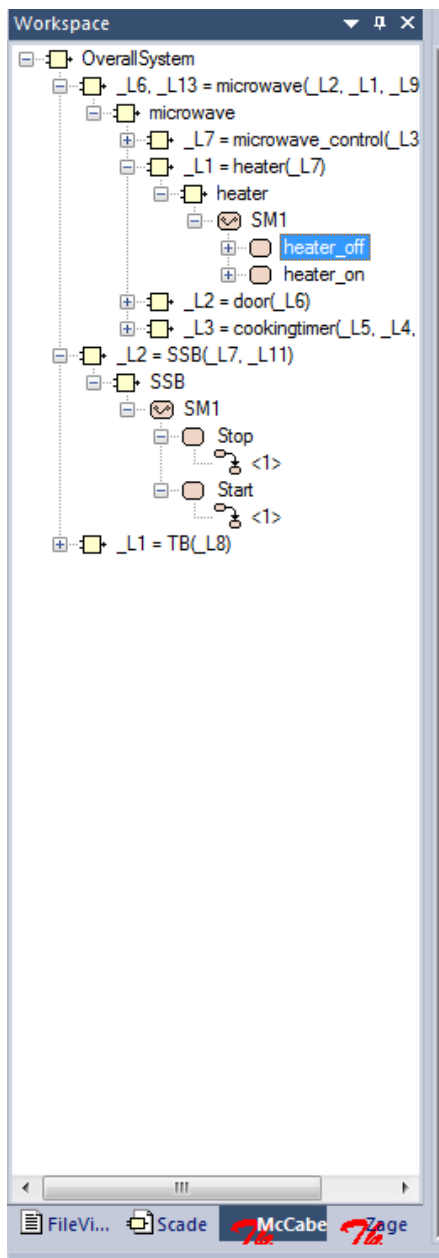
2) Another tab will be created in the “Output” area after running a tool. It will be named based on the tool that was run. These type of tabs are called “Report” in SCADE and they show a list of “interesting

points” in the model based on the analysis with additional textual notes in additional columns. Double-clicking on each row will take to the selected item in SCADE.

| Output   |       |
|--|-------|
| Script Item  | Count |
| <input checked="" type="checkbox"/> if _L5 then (_L6) else (_L9)   | 1     |
| <input checked="" type="checkbox"/> if _L15 then (_L6) else (_L4)  | 2     |
| <input checked="" type="checkbox"/> if _L13 then (_L6) else (_L10) | 3     |
| <input type="checkbox"/> heater_off                                | 5     |
| <input type="checkbox"/> heater_on                                 | 7     |
| <input type="checkbox"/> <1>                                       | 8     |
| <input type="checkbox"/> <1>                                       | 9     |
| <input type="checkbox"/> door_closed                               | 11    |

Script \ MTC \ Dump \ Build \ Simulator \ Matlab \ McCabe \ Zage \

3) A new View is created in the “Workspace” area which is generally located on the left. It is also named based on the tool that is run. This view is called a “Browser” in SCADE and shows items based on a tree. The tree is created based on the analysis that is run to give an easier understanding of the analysis result based on component relations.



When an item is double-clicked from the Report, the same item also becomes highlighted from the Browser by automatically expanding the tree. This provides a convenient way to navigate through the important items in a large model based on analysis results.

Details about understanding each result content are explained in the following sections for each tool type.

## Uninstalling TCL Scripts

While registering TCL Scripts is somewhat automatic by using a wizard, uninstalling requires the user to manually remove the registry keys in Windows. This is what it shows in the SCADE Help Topics.

[Editor](#) > [4 Customizing SCADE Suite Editor in TCL](#) > [Registering Customization Scripts](#) > [Removing Script Files from Windows Registry](#)

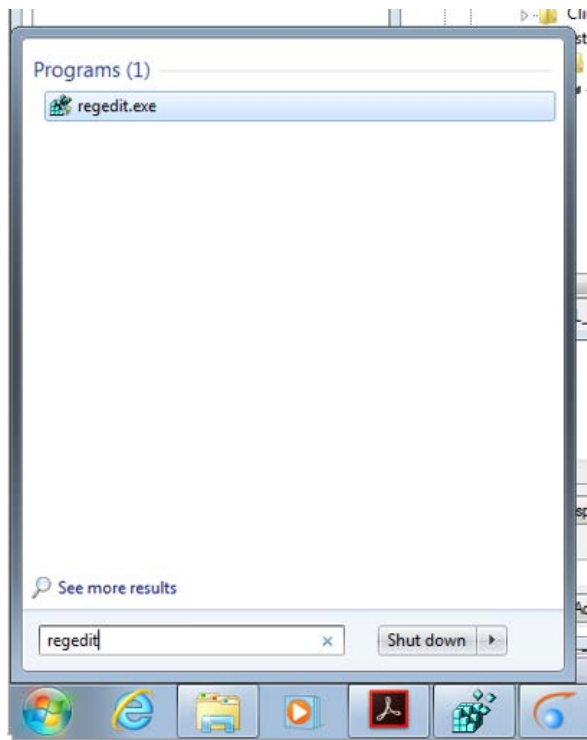
### Removing Script Files from Windows Registry

To remove custom scripts, you need to suppress manually the following registry keys:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Esterel Technologies\Studio2\Work Interfaces\<Your Custom Name>
HKEY_LOCAL_MACHINE\SOFTWARE\Esterel Technologies\Custom\Studio\Extensions\<Your Custom Name>
```

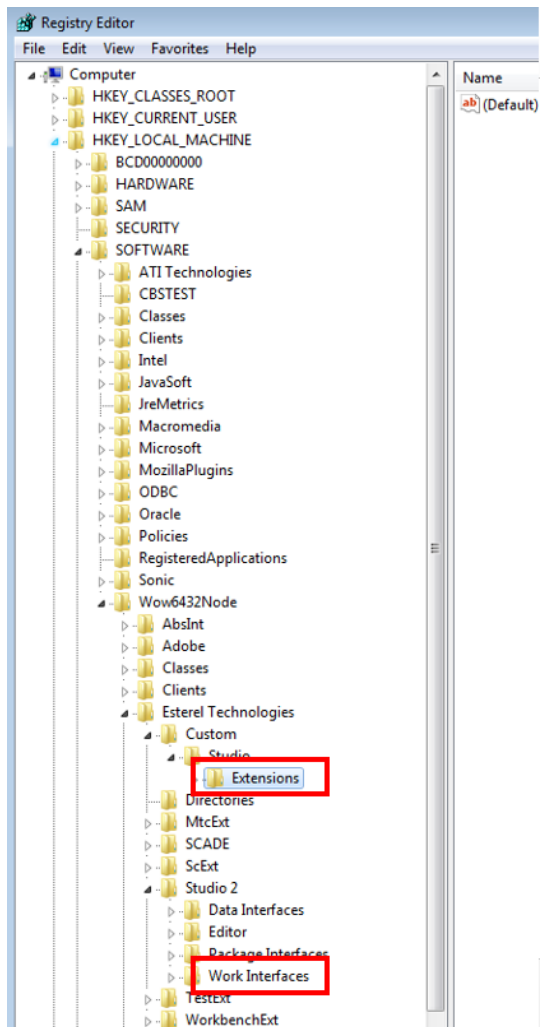
Removing registry keys require carefulness. This is why we recommended the use of a Unique Name for the TCL script during registration. SCADE already uses several TCL scripts so there will be a lot of existing registries.

Run the “regedit.exe” from the start menu.

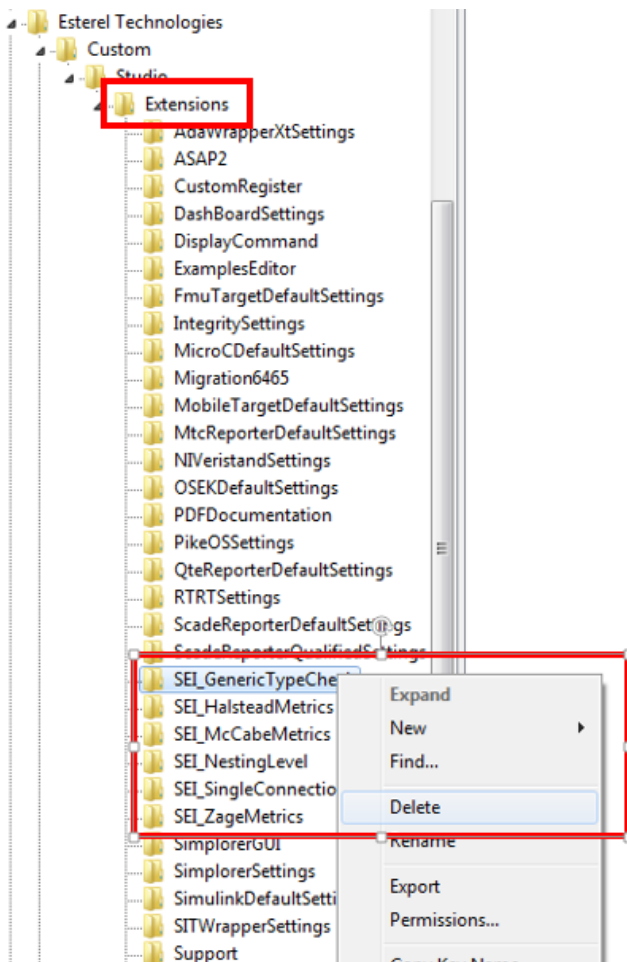


A new window name “Registry Editor” will show up. Extend the tree based on the path that you see in the Help Topic screen shot above. If you are using a 64-bit Windows, it will have an addition “WoW6432Node” between “SOFTWARE” and “Esterel Technologies.” There are two locations that you will have to check. One is in “Extensions” and the other in “Work Interfaces”





Next step is to find the TCL scrips registries that was created and deleting them. This is why we recommended to use a unique “Custom Name” during registration. These names will show up below “Extensions” and “Work Interfaces.”



In the following sections, we will explain the implementation and the result understanding on each existing tool. We will not discuss about the academic value of the tool but focus on making sure the user understands what the tool covers in the current version.

## Tool: Halstead Metrics

### Development Log

(8/18/2015) => Supports “IF Blocks” and “When Blocks” and calculates derived values.

### Background Knowledge

### Implementation Coverage

To evaluate Halstead Metrics, there are four values that we have to calculate (total number of operators and operands and total number of distinct operators and operands). To do this, we go through the model and find all “Predefined Operators,” as defined in SCADE, and count their number of instances for operators. If an operator is used at least once, it is counted as 1 for distinct number of operators. For each operator, we also investigate its inputs for operands. Total number of operands is calculated by adding all parameters that are of the type ExprId for each predefined operator instance.

To account for distinct operands, if an input to one operator is also an input to another, they count as one operand. This is calculated by first saving all local variables in an operator in a set. The total number of local variables in the set is  $N_{all}$ . Then for each suboperator, we try to remove all local variables that are used as inputs from the set. After doing this for each suboperator, the set is left with local variables that are used only for outputs or saved in local variables (never used for input to another operator). The current size of the set would be  $N_{output}$ . Then by subtracting  $N_{output}$  from  $N_{all}$ , we get the number of distinct inputs that exist for all the suboperators in an operator. We add this value for every operator.

The accuracy depends on whether the search goes to every possible place in the model that could have a predefined operator. Currently the tool starts from the selected operator and checks all equations and contained operators which is also represented as equations. If a state machine exists in the operator, it will search inside the states and the state transitions for equations and operators. For state transition, we check only the conditions. (haven't checked action). States can have another state machine inside and it will recursively search through all state machines. Search goes through each If block and their If Nodes as well as When Blocks and their When Branches.

## Result Interpretation

The Script tab will show the result of the analysis, the values for total number of operators and operands and total number of distinct operators and operands. It will also print out which operator keys (unique value in SCADE for each predefined operator) exist and how frequently they were used. Some values, such as volumn, difficulty, effort, etc. which are derived from the initial counts of operators and operands, are printed as well.

```
Iterating for distinct operators
distinctOperatorArray arraysize: 22
operator key used: 2   number of instances: 88
operator key used: 20  number of instances: 3
operator key used: 3   number of instances: 61
operator key used: 40  number of instances: 22
operator key used: 22  number of instances: 29
operator key used: 5   number of instances: 46
operator key used: 23  number of instances: 4
operator key used: 24  number of instances: 65
operator key used: 7   number of instances: 37
operator key used: 25  number of instances: 27
operator key used: 26  number of instances: 3
operator key used: 8   number of instances: 6
operator key used: 10  number of instances: 12
operator key used: 46  number of instances: 22
operator key used: 29  number of instances: 3
operator key used: 30  number of instances: 1
operator key used: 31  number of instances: 56
operator key used: 32  number of instances: 2
operator key used: 51  number of instances: 22
operator key used: 15  number of instances: 22
operator key used: 52  number of instances: 22
operator key used: 16  number of instances: 22
```

```

-----RESULT-----
total_num_operators: 575
total_num_operands: 976
num_distinct_operators: 22
num_distinct_operands: 654

program_vocabulary: 676
program_length: 1551
cal_prog_length: 6215.06551949
volumn: 14580.7640057
difficulty: 16.4159021407
effort: 239356.395053
time_to_program: 13297.577503
num_bugs: 1.28395607307

```

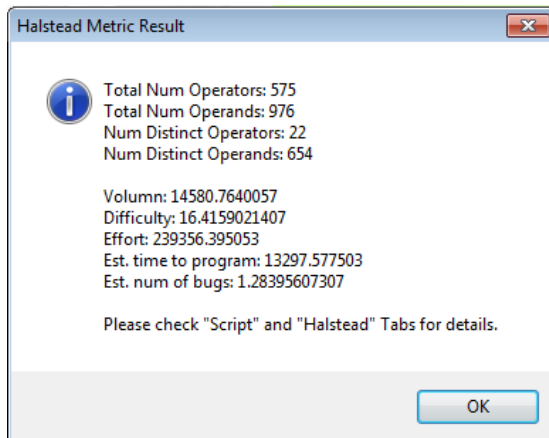
Also check Halstead tab for which predefined operators were considered and which operands were counted for total.



From the Halstead tab, user can do a visual check of which operator and operands were accounted for. The count column only shows the total count number and not the distinct number for the item.

| Output                                      |            |                  |
|---|------------|------------------|
| Script Item                                 | Count      | Type             |
| ▢ _L25                                      | 29         | Num Operand      |
| ▢ _L21                                      | Not ExprId | ForEachParameter |
| ▢ _L22                                      | Not ExprId | ForEachParameter |
| ⊞ _L21 <= _L22                              | 23         | Num Operator     |
| ▢ _L21                                      | 30         | Num Operand      |
| ▢ _L22                                      | 31         | Num Operand      |
| ⊞ _L1 >= _L2                                | 24         | Num Operator     |
| ▢ _L1                                       | 32         | Num Operand      |
| ▢ _L2                                       | 33         | Num Operand      |
| ⊞ if _L3 then (_L1) else (_L2)              | 25         | Num Operator     |
| ▢ _L3                                       | 34         | Num Operand      |
| ▢ _L1                                       | Not ExprId | ForEachParameter |
| ▢ _L2                                       | Not ExprId | ForEachParameter |
| ⊞ physical_push or heater_state = heaterOff | 26         | Num Operator     |
| ⇒ physical_push                             | 35         | Num Operand      |
| ⊞ heater_state = heaterOff                  | Not ExprId | ForEachParameter |

When the analysis finishes, a pop up window would also show a summary of the analysis.



## Tool: McCabeMetrics

### Development Log

(8/18/2015) => Supports "IF Blocks" and "When Blocks"

### Background Knowledge

#### Implementation Coverage

For McCabe metrics, we search for IF and CASE statements and Loops (FOR statements) in the model. These are counted to give the McCabe Complexity value. To do this, the tool currently goes through the model and looks for IF and CASE operators and count them accordingly. For CASE statements, it will consider the number of case values and whether this is default or not. "If Blocks" "If Node" "When Block" and "When Branch" are counted accordingly. State machines generate case statement in code. Thus, the number of states are considered in the tool. Also, it was recognized that if a state machine has at least one Strong transition, each state counts for two cases. This is also considered. Each transition in a state machine generates one if statement. Among the predefined operators, we found that the operators in the "Higher Order" category generates FOR loops. Each FOR loop is counted as 1.

Currently the tool starts from the selected operator and checks all equations and contained operators which is also represented as equations. If a state machine exists in the operator, it will search inside the states and the state transitions for the aforementioned elements. For state transition, we check only the conditions. (haven't checked action). States can have another state machine inside and it will recursively search through all state machines. Search goes through each If block and their If Nodes as well as When Blocks and their When Branches.

#### Result Interpretation

After running the tool, the final McCabe complexity value is printed in the Sript tab in Output.

```

Output
-----ForEachEquation2 Entry: Equation>>>...
>>>_L12 = heaterOff
-----ForEachEquation2 Entry: Equation>>>...
>>>test_remainingTime = _L13

Iterating for distinct operators
distinctOperatorArray arraysize: 9
operator key used: 22  number of instances: 1
operator key used: 8  number of instances: 1
operator key used: 31 number of instances: 9
operator key used: 26 number of instances: 2
operator key used: 23 number of instances: 1
operator key used: 24 number of instances: 3
operator key used: 7  number of instances: 1
operator key used: 21 number of instances: 2
operator key used: 29 number of instances: 2
















-----RESULT-----
total_num_operators: 22
num_distinct_operators: 9

McCabe_complexity: 27
Also check McCabe tab to see which items contributed to complexity values

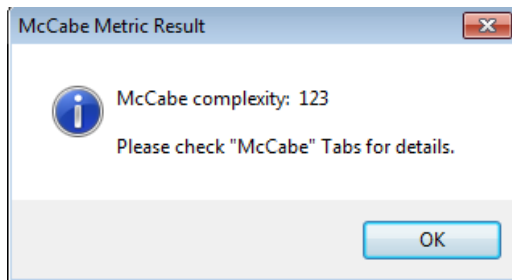
```

Navigation bar: Messages | Script | MTC | Dump | Build | Simulator | Matlab | Halstead | McCabe

The McCabe tab shows which elements contributed to the final complexity count.

| Output  |       |
|---|-------|
| Script Item   | Count |
|  <1>                             | 9     |
|  door_closed                     | 11    |
|  door_open                       | 13    |
|  <1>                             | 14    |
|  <1>                             | 15    |
|  if _L20 then (_L15) else (_L21) | 16    |
|  if _L25 then (_L29) else (_L28) | 17    |
|  if _L30 then (_L32) else (_L31) | 18    |
|  if _L25 then (_L21) else (_L22) | 19    |
|  if _L3 then (_L1) else (_L2)    | 20    |
|  Stop                            | 22    |
|  Start                           | 24    |
|  <1>                             | 25    |
|  <1>                             | 26    |
|  if _L2 then (_L5) else (_L4)    | 27    |

Navigation bar: Messages | Script | MTC | Dump | Build | Simulator | Matlab | Halstead | McCabe



## Tool: ZageMetrics

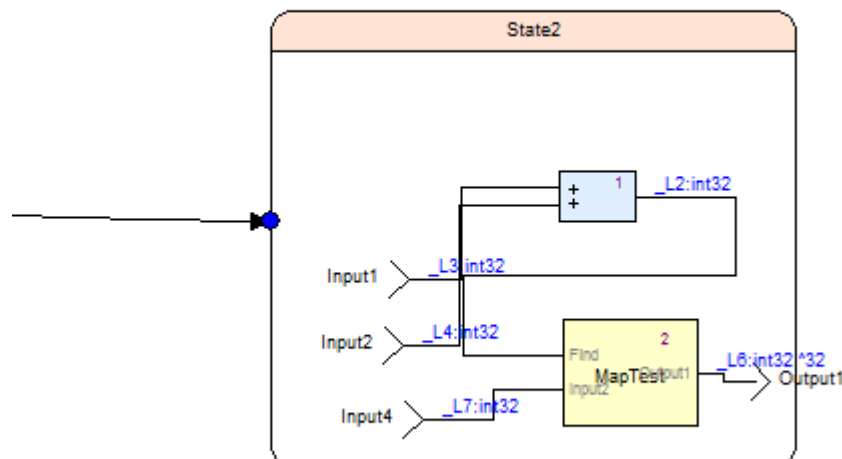
### Development Log

(8/18/2015) => Supports "IF Blocks" and "When Blocks" and gives average values.

### Background Knowledge

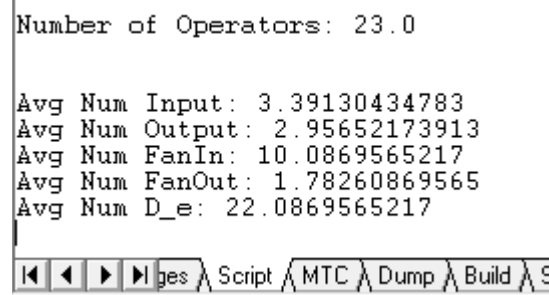
### Implementation Coverage

For Zage Metrics, we look for four values (Number of input/output and Number of Fan-In and Fan-Out). We give this value for each Non-predefined Operator. Number of input and output is counted by analyzing the equation where the operator is called. Fan-In is counted by adding the number of times an operator type is used in the model. Fan-Out is counted by the number suboperators that an operator has inside directly. It will analyze for operators that given in the SCADE as libraries to use, ex) MAX, MIN. It will bypass state machines, but count any suboperator that is inside the state machine such as the picture below and consider it as direct.



Result is mainly shown at the end of the tab named “Zage.” In the end (scroll down to the end of the table), it lists all the Non-predefined operators that were used and show the four values and the D\_e value (external design metric) which is currently the multiple of Num Input and Num Output summed with the multiple of FanIn and FanOut. Weight factors are simplified to 1. Before this listing, it shows how inputs and outputs of the operators are counted.

When the analysis finishes, a pop up window also shows the average of these values.





## Tool: Nesting Level Check

### Development Log

(8/18/2015) => Supports "IF Blocks" and "When Blocks"
















### Background Knowledge


### Implementation Coverage

Nesting Level Check goes through modeling components and counts the depth of Non-predefined operators and state machine states.

### Result Interpretation

The Script tab shows the maximum nesting level and the NestLevel tab shows the level of each Non-predefined operators and states that were identified in the model.

| Output  |       |  |
|---|-------|--|
| Script Item   | Level |  |
|  microwave         | 1     |  |
|  microwave_control | 2     |  |
|  heater            | 2     |  |
|  heater_off       | 3     |  |
|  heater_on       | 3     |  |
|  door            | 2     |  |
|  door_closed     | 3     |  |
|  door_open       | 3     |  |
|  cookingtimer    | 2     |  |
|  Min             | 3     |  |
|  Max             | 3     |  |
|  SSB             | 1     |  |
|  Stop            | 2     |  |
|  Start           | 2     |  |
|  TB              | 1     |  |

 MTC \ Dump \ Build \ Simulator \ Matlab \ Halstead \ McCabe \ Zage \ NestLevel \

## Tool: Generic Type Check

### Development Log

(8/18/2015) => Supports "IF Blocks" and "When Blocks"

## Background Knowledge

### Implementation Coverage

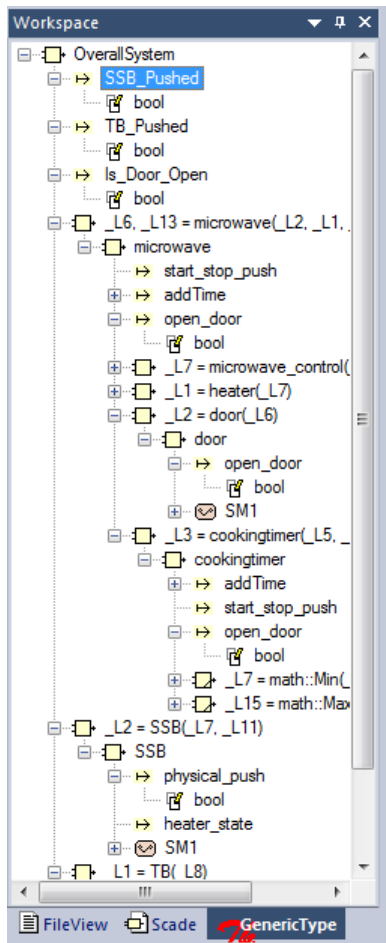
The Generic Type Check tool looks for usages of integer and boolean type variables in undesirable places. Current version only goes through all operators that are used from the selected operator when running the tool. If the input variable has a type of bool or int\*\* or uint\*\*, it will be reported. Tool should be more sophisticated to find hints of integers used as.

### Result Interpretation

Input variables will be shown in the GenericType tab in Output. Beneath the input variable, it will show the variable type. Double clicking on the variable will show the location of the variable in the GenericType tab in the Workspace.

| Output           |       |      |
|------------------|-------|------|
| Script Item      | Count | Type |
| ⇒ SSB_Pushed     |       |      |
| ⇒ bool           |       |      |
| ⇒ TB_Pushed      |       |      |
| ⇒ bool           |       |      |
| ⇒ Is_Door_Open   |       |      |
| ⇒ bool           |       |      |
| ⇒ addTime        |       |      |
| ⇒ uint16         |       |      |
| ⇒ open_door      |       |      |
| ⇒ bool           |       |      |
| ⇒ remaining_time |       |      |
| ⇒ uint16         |       |      |
| ⇒ open_door      |       |      |
| ⇒ bool           |       |      |

Messages | Script | MTC | Dump | Build | Simulator | Matlab | GenericType |



## Tool: Single Connection Check

### Development Log

(8/18/2015) => Supports "IF Blocks" and "When Blocks"

### Background Knowledge

### Implementation Coverage

Single Connection Check tool look for operators that may be better to be merged with other operators because it consists of a single input connection.

### Result Interpretation

A dialog appears with the total number of operators that have single connections. All Non-predefined operators in the model are shown in the Connection tab from Output. Below each operator, it shows the counting of the inputs and indicates whether there is only one connection.

| Output                         |                     |  |
|--------------------------------|---------------------|--|
| Script Item                    | Note                |  |
| ▢ _L6                          | 2                   |  |
| ▢ _L15 = math::Max(_L12, _L16) |                     |  |
| ▢ math::Max(_L12, _L16)        |                     |  |
| ▢ _L12                         | 1                   |  |
| ▢ _L16                         | 2                   |  |
| ▢ _L2 = SSB(_L7, _L11)         |                     |  |
| ▢ SSB(_L7, _L11)               |                     |  |
| ▢ _L7                          | 1                   |  |
| ▢ _L11                         | 2                   |  |
| ▢ _L1 = TB(_L8)                |                     |  |
| ▢ TB(_L8)                      |                     |  |
| ▢ _L8                          | 1                   |  |
| ▢ TB                           | Candidate For Merge |  |

## Tool: Data Flow

### Development Log

(9/22/2015) => Newly added

### Background Knowledge

This tool is implemented to let users easily follow the effect of input data. When selecting an operator and running the tool, it would start with the inputs of the operator and create a rather long tree of how the data is broadcasted. However, the path does not only depend on the value of the input but also on the effect of the input. For example, if an input is used on the transition of a state machine, the data path would continue in the target state. If the input is used as an if-else condition, data path is split into the “then” and “else” statements. The correct term for the tool would be “data dependency.”

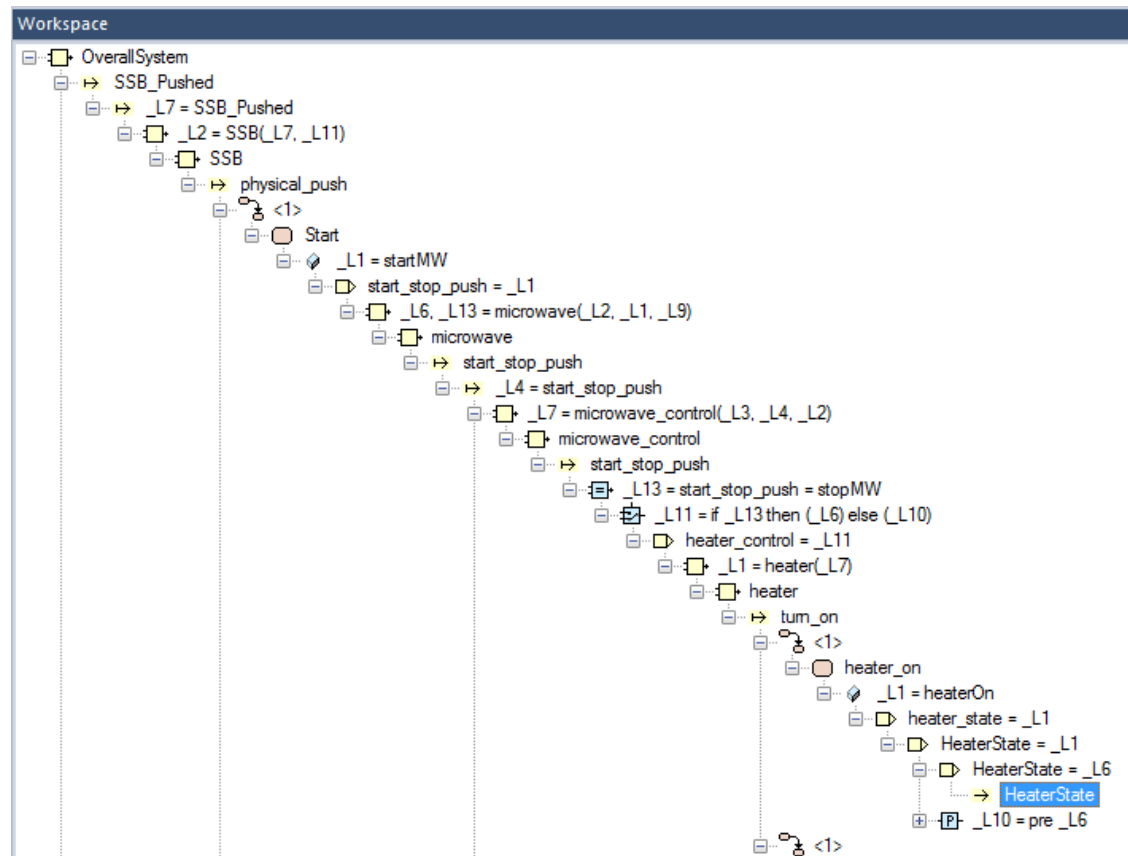
### Implementation Coverage

Current implementation covers most of the data paths that are possible in Scade. Since, the tool is extensive, when running the tool it may feel like the tool is hanging but if you wait more, it is likely to finish. Identifying cyclic dependency is a main issue for creating the data path in a tree based manner. There was a couple of ways to identify possible existence of cycles. The ones we used to stop the data path were the usage of the PREVIOUS and FOLLOWED BY operators and the LAST variable component. When they are identified to be called more than once for the same input in the same location, the data path is cut short. Each data path continues until it identifies a cycle condition or until it reaches an output (especially the output of the operator that was selected to run the analysis). Currently we do not create data paths for constants, which could be useful in the future.

## Result Interpretation

Result is mainly shown in the tabs named "DataFlow." From the tab in the Output, you will see a list of all the operators that was called for each input and later which output that it has reached, unless it was terminated by the cycle condition. By clicking on one of the outputs or terminating points, you can easily extend the tree to show the whole path. We have not added any numerical values to the paths yet.

| Output                                 |                                   |               |
|--|-----------------------------------|---------------|
| Script Item                            | Comment                           | Current Input |
| ⇒ SSB_Pushed                           | Starting INPUT in SSB_Pushed-->>> | SSB_Pushed    |
| ⇒ _L2 = SSB(_L7, _L11)                 | Called Operator                   | SSB_Pushed    |
| ⇒ _L6, _L13 = microwave(_L2, _L1, ...) | Called Operator                   | SSB_Pushed    |
| ⇒ _L7 = microwave_control(_L3, ...)    | Called Operator                   | SSB_Pushed    |
| ⇒ _L1 = heater(_L7)                    | Called Operator                   | SSB_Pushed    |
| ⇒ HeaterState                          | -->>>End OUTPUT in OverallSystem  | SSB_Pushed    |
| ⇒ _L2 = SSB(_L7, _L11)                 | Called Operator                   | SSB_Pushed    |
| ⇒ HeaterState                          | -->>>End OUTPUT in OverallSystem  | SSB_Pushed    |
| ⇒ _L10 = pre_L6                        | Terminating due to possible Cycle | SSB_Pushed    |
| ⇒ _L3 = cookingtimer(_L5, _L4, _L6)    | Called Operator                   | SSB_Pushed    |
| ⇒ _L15 = math::Max(_L12, _L16)         | Called Generic Operator           | SSB_Pushed    |
| ⇒ _L7 = math::Min(_L2, _L6)            | Called Generic Operator           | SSB_Pushed    |
| ⇒ _L15 = math::Max(_L12, _L16)         | Called Generic Operator           | SSB_Pushed    |
| ⇒ _L17 = pre_L3                        | Terminating due to possible Cycle | SSB_Pushed    |



Copyright 2015 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

This report was prepared for the  
SEI Administrative Agent  
AFLCMC/PZM  
20 Schilling Circle, Bldg 1305, 3rd floor  
Hanscom AFB, MA 01731-2125

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This material has been approved for public release and unlimited distribution except as restricted below.

Internal use:\* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use:\* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

\* These restrictions do not apply to U.S. government entities.

Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM-0002574