ENTERPRISE ARCHITECT

User Guide Series

Profiling

Author: Sparx Systems

Date: 2021-09-02

Version: 15.2

Table of Contents

Profiling

During the lifetime of software applications, it is not uncommon to investigate application tasks that are determined to be performing slower than expected. You might also simply want to know what is going on when you '*press this button*'! You can work this out quite quickly in Enterprise Architect by using its **Profiler**. Results can usually be produced in a few seconds and you will quickly be able to see the actions that are consuming the application and the functions involved. In the Execution Analyzer, The feature employs two separate strategies; *Process Sampling* and *Process Hooking*. In one, samples are taken at regular intervals to identify CPU-intensive patterns, while in the other, the process is hooked to record demands made on memory. Data is analyzed to produce a weighted Call Graph. Behaviors are usually identifiable as root nodes (entrypoints) in the graph, or branches near these points. All reports can be reviewed on demand. They can be saved to file within the model, both as Artifact elements and as **Team Library** posts.

Access

Ribbon	Execute > Tools > **Profiler**
Other	Execution Analyzer toolbar : Analyzer Windows | **Profiler**

Call Sampling
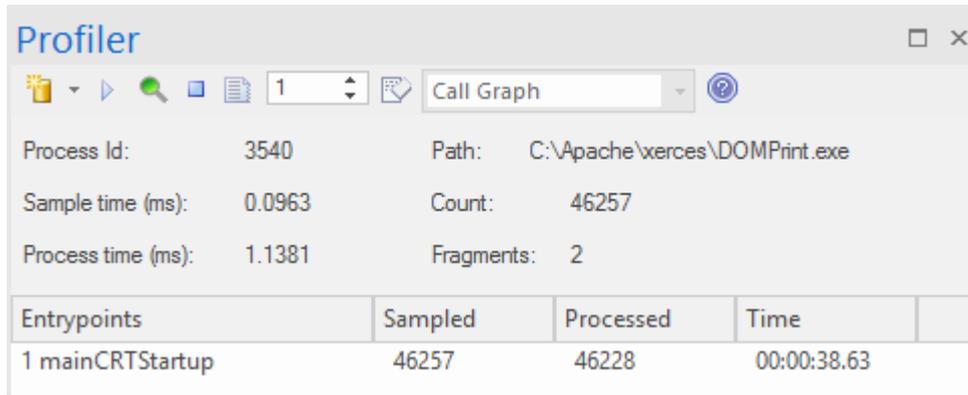
The **Profiler** is controlled using its toolbar buttons. Here you can attach the Profiler to an existing process (or JVM), or launch the application for the active Analyzer Script. The **Profiler window** displays the details of the target process as it is profiled. These details provide feedback, letting you see the number of samples taken. You also have options for pausing and resuming capture, clearing captured data and generating reports. You can gain access to the reporting feature by pausing the capture - the reporting feature is disabled whilst data capture is in progress.

Weighted Call Graph
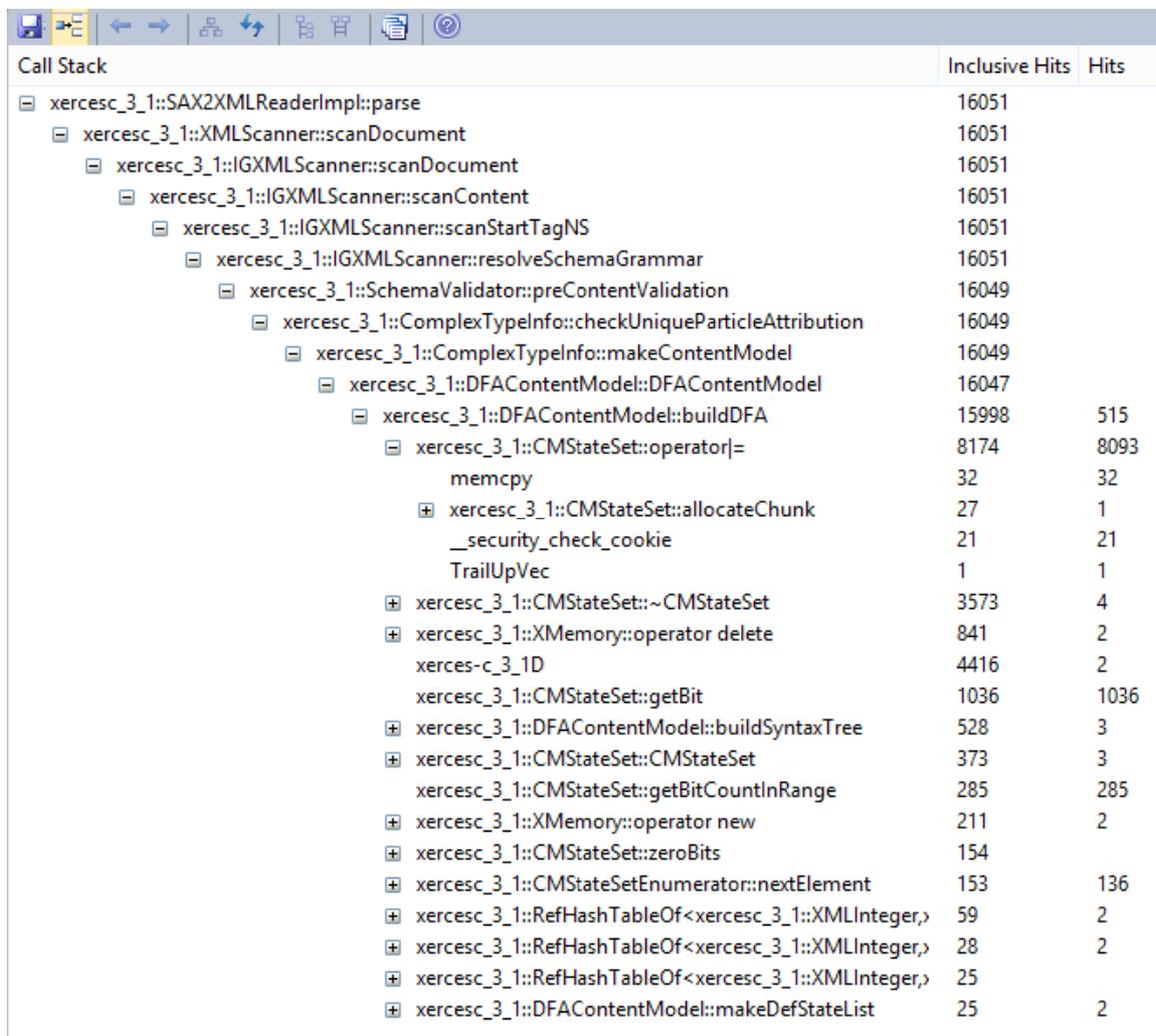
This detailed report shows the unique set of Call Stacks/behaviors as a weighted Call Graph. The weight of each branch

is depicted by a hit count, which is the total hits of that branch plus all branches from this point. By following the hit trail, a user can quickly identify the areas of code that occupied the program the most during the capture period.

Stack Profile
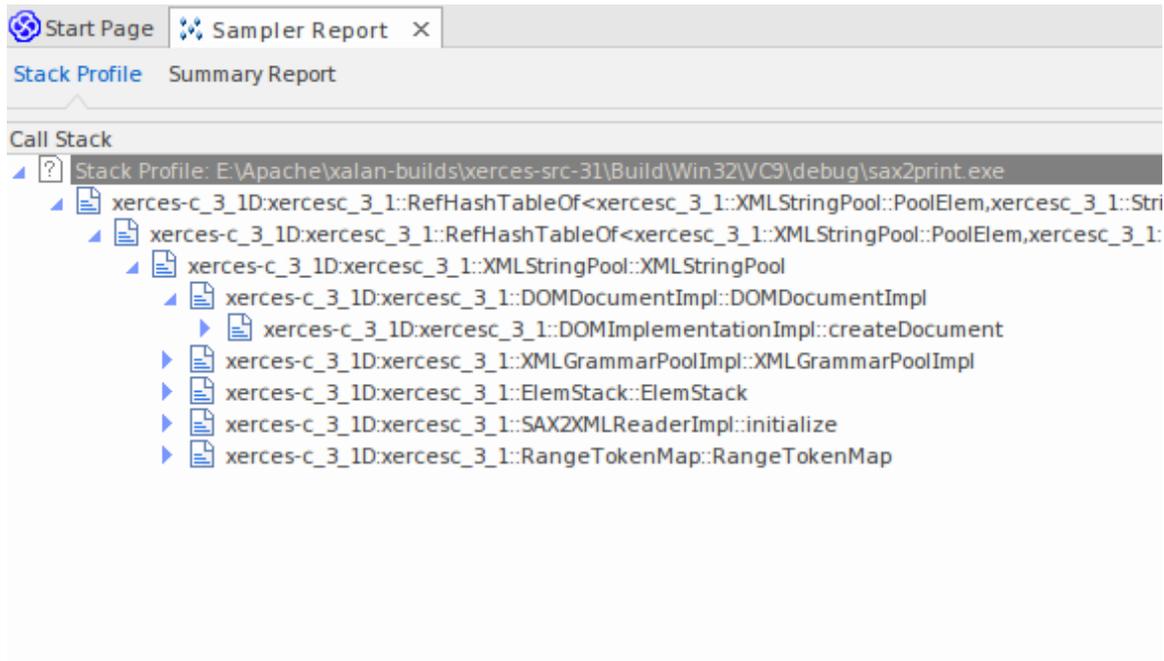
Stack Profiles are a taken to discover the different ways (stacks) and the count of ways that a particular function is invoked during the running of the program. Unlike other the other profiler modes, this profile is activated through the use of a **Profile Point**, which is a special kind of breakpoint marker. The marker is set in the source code like any other breakpoint. When the breakpoint is encountered by the program, the stack is captured. When you later produce the report, the stacks are analyzed and a weighted call graph produced. The graph shows the unique stacks that were involved in that function during the time the profiler was running, The 'Hit Count' column indicates the count of times that same stack occurred.

```
106
107 template <class TVal, class THasher>
108 void RefHashTableOf<TVal, THasher>::initialize(const XMLSize_t modulus)
109 {
110     if (modulus == 0)
111         ThrowXMLwithMemMgr(IllegalArgumentException, XMLExcepts::HshTbl_ZeroMo
112
113     // Allocate the bucket list and zero them
114     fBucketList = (RefHashTableBucketElem<TVal>**) fMemoryManager->allocate
115     (
116         fHashModulus * sizeof(RefHashTableBucketElem<TVal>*)
117     );
118     for (XMLSize_t index = 0; index < fHashModulus; index++)
119         fBucketList[index] = 0;
120 }
121
```

Memory Profiles
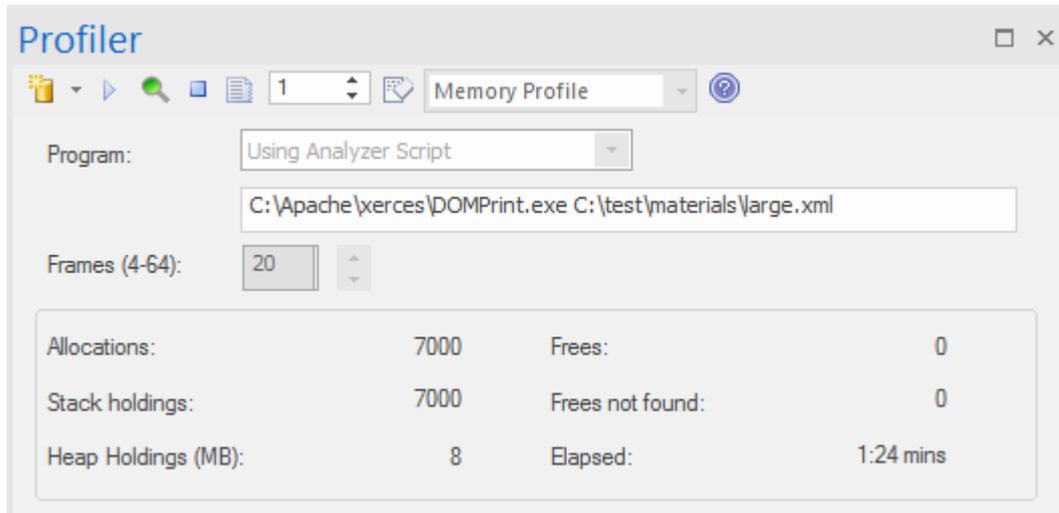
The Memory profile tracks allocations, ignoring when memory is freed. It uses this information to rate the executing code's demands for memory, in terms not of the amount of memory but of the frequency of demands. The *Allocations* figure is the total number of memory allocations requested. The *Stack Holdings* is the number of stack traces taken at those times, and the *Heap Holding* figure is the total amount of memory obtained by these calls. Note that profiling can be turned on and off on demand. There is also no need to rebuild your program to get it to work as there is no linkage involved.

Memory Graph
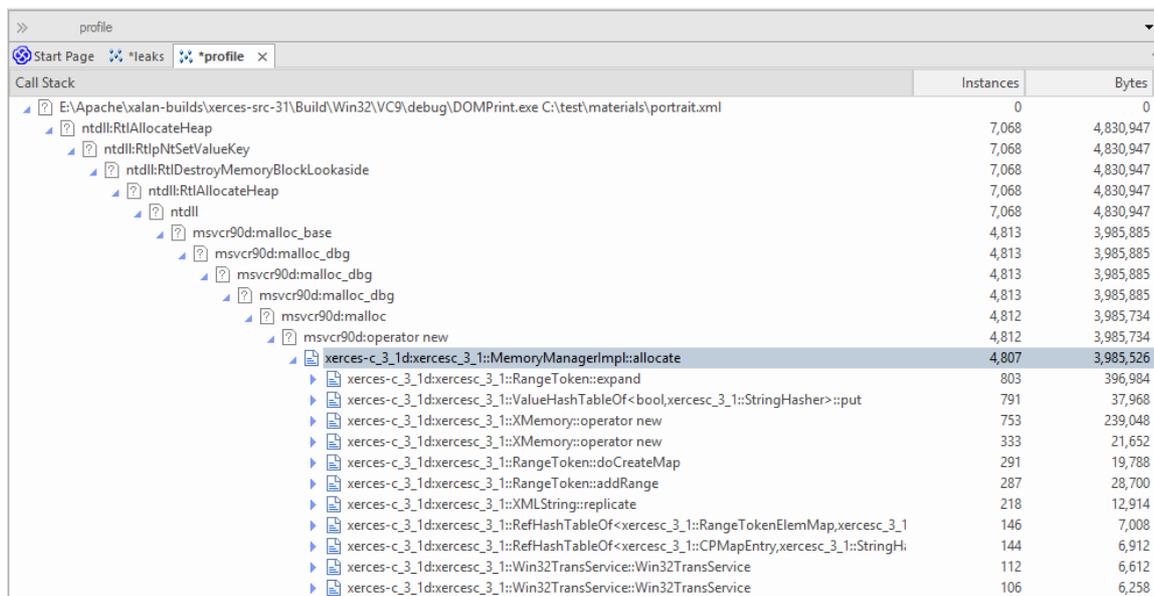
This example is of a report produced from **Profiling** a demonstration program in the Xerces project from Apache. The program iterates over the Document **Object Model** (DOM) for a provided XML file.

Function Summary Report

Name	Inclusive Hits
profiler/Example.Run	156
profiler/Example.main	156
java/io/FileOutputStream.write	154
java/io/PrintStream.printIn	154
profiler/Example.Print	154
profiler/Example.MakeItalianCars	2
profiler/Example.NewCar	2

This summary report lists the functions and only those functions executed during the sample period. Functions are listed by total invocations, with a function that presents twice in separate Call Stacks appearing before a function that appears just the once.

Function Line Report

LineNo	Hits	Code
54	1	for(int n = 0; n < 10000; n++)
55		{
56	1408	m_Cars = new Collection<Car>();
57	1408	if((n % 3)>0)
58		{
59	938	for(int i = 0; i < 1000; i++)
60		{
61	938000	MakeItalianCars();
62		}

This detailed report shows the source code for a function line by line displaying beside it the total times each was executed. We uncovered code using this report, that exposed case statements in code that never appeared to be executed.

Support

The **Profiler** is supported for programs written in C, C++, Visual Basic, Java and the Microsoft .NET languages. Memory profiling is currently available for native C and C++ programs.

Notes

- The **Profiler** is available in the Enterprise Architect Professional Edition and above
- The Profiler can also be used under WINE (Linux and Mac) for **Profiling** standard Windows applications deployed in a WINE environment

System Requirements

Using the **Profiler**, you can analyze applications built for these platforms:

- Microsoft ™ Native (C++, C, Visual basic)
- Microsoft .NET (supporting a mix of managed and unmanaged code)
- Java

Microsoft Native applications

For C, C++ or Visual Basic applications, the **Profiler** requires that the applications are compiled with the Microsoft ™ Native compiler and that for each application or module of interest, a PDB file is available. The Profiler can sample both debug and release configurations of an application, provided that the PDB file for each executable exists and is up to date.

Microsoft .NET applications

For Microsoft .NET applications, the **Profiler** requires that the appropriate Microsoft .NET framework is installed, and that for each application or module to be analyzed, a PDB file is available.

Java

For Java, the **Profiler** requires that the appropriate JDK from Oracle is installed.

The classes of interest should also have been compiled with debug information. For example: "java -g *.java"

- New instance of application VM is launched from Enterprise Architect - no other action is required
- Existing application VM is attached to from within Enterprise Architect - the target Java Virtual Machine has to have been launched with the Enterprise Architect profiling agent

These are examples of command lines to create a Java VM with a specific **JVMTI** agent:

1. java.exe -cp "%classpath%;.\" -agentpath:"C:\Program Files (x86)\Sparx Systems\EA\vea\x86\ssamplerlib32" myapp

2. java.exe -cp "%classpath%;.\" -agentpath:"C:\Program Files (x86)\Sparx Systems\EA\vea\x64\ssamplerlib64" myapp

(Refer to the JDK documentation for details of the -agentpath VM startup option.)

Getting Started

The **Profiler** can be used to investigate performance issues, providing four separate tools for you to choose from, namely:

- Call Graph
- Stack Profile
- Memory Profile
- Memory Leaks

You select these tools from the Profiler toolbar.

Access

Ribbon	Execute > Tools > **Profiler**

Tools

| Tool	Description
Call Graph	Analyzes performance by taking samples during an activity in a program. Each sample represents a stack. The samples are taken at intervals controlled using the toolbar. In this scenario, poor performance is rated by the patterns of behavior that repeat the most during the sample time period. This figure is used to weight the Call Graph produced.
Memory Profile	Analyzes performance by hooking the memory allocations made by a program. In this scenario, poor performance is rated by the activities making the most requests for memory. This figure is used to weight the Call Graph produced.
Stack Profile	The Stack **Profiler** enables you to set a marker in your source code so that whenever execution hits that marker, a full stack trace is captured. As the application continues executing and the marked position is accessed from a variety of places within the running executable, a very detailed and useful picture is built up showing hot spots and usage scenarios for a particular point in code. The Stack Profile report, like the Memory Profile report, is displayed in 'reverse stack' order. This means that the root of the report is always a single node (in this case the marker) and the tree then fans out to show all the various places the marked position has been accessed from.
Memory Leaks	Analyzes memory leaks by hooking the memory operations performed by a program. What is produced is a Call Graph presenting the Call Stacks that allocated memory for which a free operation was not detected.

Toolbar Buttons

Button	Action
	Displays a menu of options for managing your **Profiling** session.
	Launches the configured application to be profiled. By default, this is the application configured in the active Analyzer Script.
	Indicates the state of the sampler. When green, sampling is enabled; when red, sampling is disabled.
	Stops the **Profiler** process; if any samples have been collected, the **Report button** and **Discard Data button** are active.
	Generates a report from the current data collection.
	Displays the **Profiling** tool in use, which determines the fields shown in the **Profiler window**. Click on the **drop-down arrow** and select a different tool, which changes the window fields.
	Discards the collected data. You are prompted to confirm the discard.
	Displays the Help topic for this window.

Call Graph

- Quickly discover what a program is doing at any point in time
- Easily identify performance issues
- Be surprised how quickly you can realize improvements
- See your improvements at work and have the evidence
- Support for C/C++, .NET and Java platforms

Usage

The 'Call Graph' option is typically used in situations where an activity is performing slower than expected, but it can also be used simply to better understand the patterns of behavior at play during an activity.

Operation

The **Profiler** operates by taking samples - or Call Stacks - at regular intervals over a period of time; the interval is set using the Profiler toolbar. You use the Profiler to run a particular program, or you can attach to an existing process. The Profiler capture is controlled, and you can pause and resume capture at any time. You can also elect to have capture initiated immediately when the Profiler is started. If necessary, you can discard any captured samples and start again during the same session. If you cannot continue with the same session, restarting the Profiler is quick and easy.

Results

Results can be produced at any time during the session; however, capture must be disabled in order for the **Report button** to become active. It is up to you to decide how long you let the **Profiler** run. You might know when an activity is finished, or it might be apparent for other reasons. The reason you are here might be that an activity is not completing at all.

You enable the Report button by either pausing capture or stopping the Profiler altogether.

Results are displayed in a Report view. The report opens with two tabs initially visible: the Call Graph and the Function Summary. The reports can be saved to file, stored in the model as Artifacts or posted in the **Team Library**.

Stack Profile

The Stack **Profiler** enables you to set a marker in your source code so that whenever execution hits that marker, a full stack trace is captured. As the application continues executing and the marked position is accessed from a variety of places within the running executable, a very detailed and useful picture is built up showing hot spots and usage scenarios for a particular point in code.

The Stack Profile report, like the Memory Profile report, is displayed in 'reverse stack' order. This means that the root of the report is always a single node (in this case the marker) and the tree then fans out to show all the various places the marked position has been accessed from.

Usage

Use the Stack Profile mode to produce a report that shows the unique ways in which a function can be invoked during the running of a program. Determine the parts of the model that rely on this function and their frequency.

Operation

Profiler modes are selected using the Profiler control Toolbar. If a Profiler Point is already created, it is displayed. The Profiler Point is the point at which stack traces are captured. You can set the Profiler Point using the Set button on the control itself, once the mode is selected. After deciding on the **Profile Point**, build the project to be sure everything is up to date, then start the Profiler. The number of unique stack holdings detected is visible during the run.

Results

A results can be produced by clicking the report button on the **Profiler** control Toolbar. This button is enabled when either:

- Capture is turned off (using the Pause Button) or
- The Profiler is stopped (using the Stop Button)

The results produced are displayed as a weighted call graph, where the lines on the graph represent a unique stack, and weighted to show the higher frequency stacks first. The report can then be saved, either to file or to the model, using the context menu of the report itself.

Memory Profile

- Quickly rate performance of activities that interest you
- Nothing influences a discussion more than evidence
- Reward your efforts by working in those areas that will make a difference
- Surprise yourself by delivering optimizations you might not have known existed

Usage

The Memory Profile can be used to reveal how activities perform in regard to memory consumption. Using this mode, a user would be interested in questioning the frequency of demands made for memory during a task. They would be less interested in the actual amount consumed. A well managed activity might make relatively few calls to allocate resources but allocate enough memory to do its job efficiently. Other activities might make many thousands of requests, and that typically makes them less efficient. This mode is useful for detecting those scenarios.

Operation

The Memory Profile works by hooking the process in question, so that program has to be launched using the tool in Enterprise Architect. Unlike the Call Graph option, you cannot attach to an existing process. When the program is started, hooking mechanisms track the allocation of memory; this information is collected and collated in Enterprise Architect. You can easily monitor the number of allocations being made. Also, the process is controlled; that is, the memory hooks can be turned on and off on demand. If you might have mistimed some action, you can pause capture, discard the data and resume capture again easily.

Results

Results can be produced at any time during the session; however, capture must be disabled in order for the **Report button** to become active. It is your decision how long you let the **Profiler** run. You enable the Report button by either pausing capture or stopping the Profiler altogether.

Results are displayed in a Report view. The report initially opens with two tabs visible; a single weighted Call Graph and a Function Summary. The Call Graph depicts all the Call Stacks that led to memory allocations, which are aggregated and weighted according to the frequency of the pattern.

Requirements

For best results, the image and its modules should be built with debug information included, and without optimizations. Any module with the Frame Pointer Omission (FPO) optimization is likely to produce misleading results.

Memory Leaks
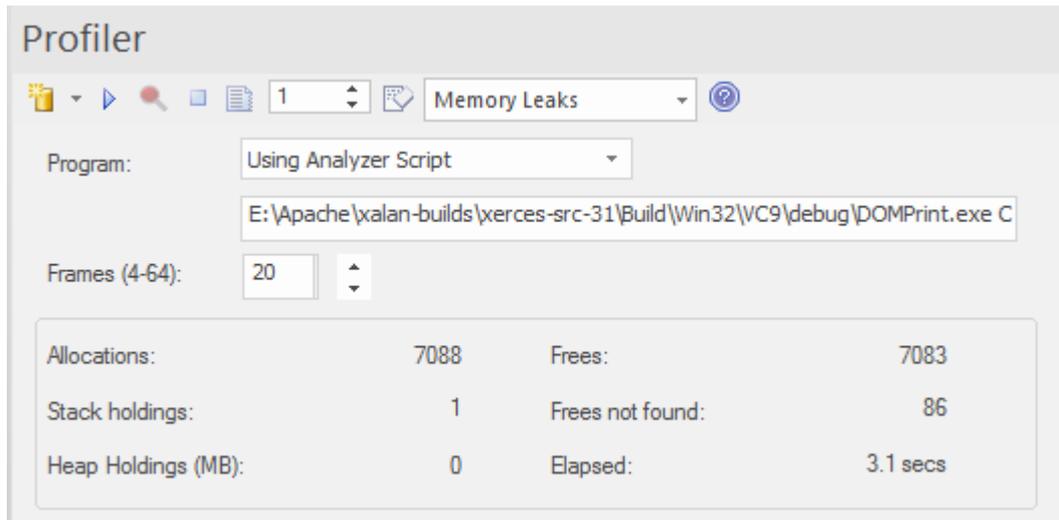
The **Profiler** control, showing the count of memory allocations and the count of operations that are memory free.
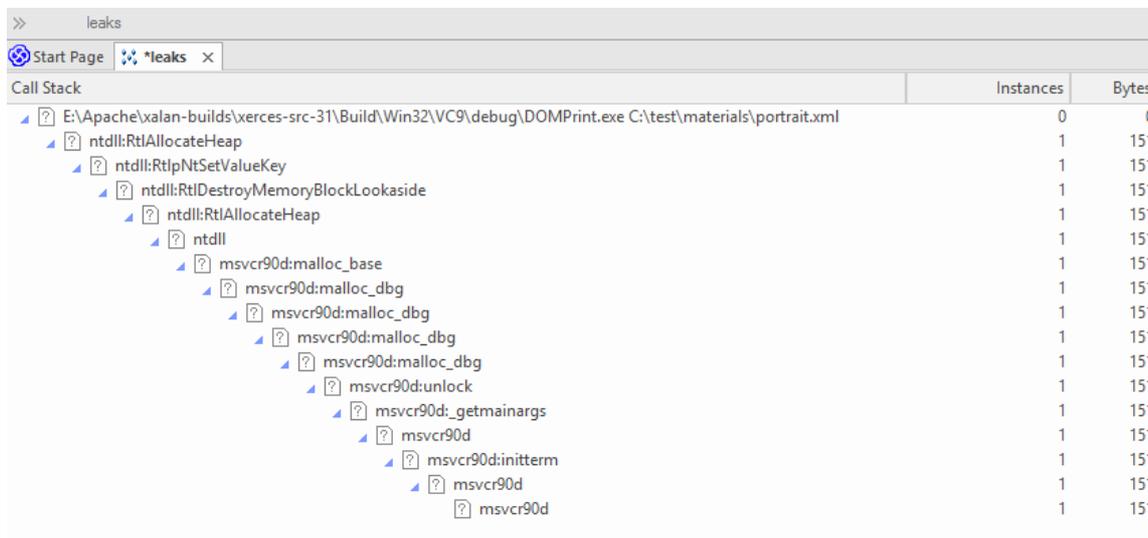
A well behaved program.

Memory leak detection is a road well traveled. Although many other good options are available, we believe our approach has major benefits, such as:

- No changes at all to existing project build
- No header files required by the project code
- No runtime dependencies to worry about
- No system configuration to think about

Usage

A person would use this mode to track memory leaks in an application or in an activity within the application. A memory leak from the **Profiler**'s point of view is a successful call made to a memory allocation function that returns a memory address for which no matching call is made to free that address.

Operation

The Memory Leak detection works through hooking. The memory routines of the process are hooked to track when memory is both allocated and freed. Call Stacks are captured at the point of the allocation and this information is collated in Enterprise Architect to produce a report in the form of a Call Graph. Capture is controlled; that is, the hooking mechanisms can be enabled or disabled on demand.

Depending on the type of program and its memory consumption, you could employ an appropriate strategy. For small programs, you might track the program from start to finish. For larger windowed programs, you would probably do better by toggling capturing before and after a specific task to avoid tracking too much data.

Results

Results can be produced at any time during the session; however, capture must be disabled in order for the **Report button** to become active. It is your decision how long you let the **Profiler** run. You enable the Report button by either pausing capture or stopping the Profiler altogether.

Results are displayed in a Report view. The report initially opens with two tabs visible; a single weighted Call Graph and a Function Summary. The Call Graph depicts all the Call Stacks that led to memory allocations, and are aggregated and weighted according to the frequency of the pattern.

Reports can contain a variable amount of 'noise'. To focus on an area you have specific concerns for, locate a function known to you in the summary report and use that to navigate directly into the line in the graph where it is featured.

Requirements

For best results, the image and its modules should be built with debug information included, and without optimizations. Any module with the Frame Pointer Omission (FPO) optimization is likely to produce misleading results.

Setting Options

The first icon on the **Profiler window** toolbar displays a list of options that you can set to tailor your **Profiling** session.
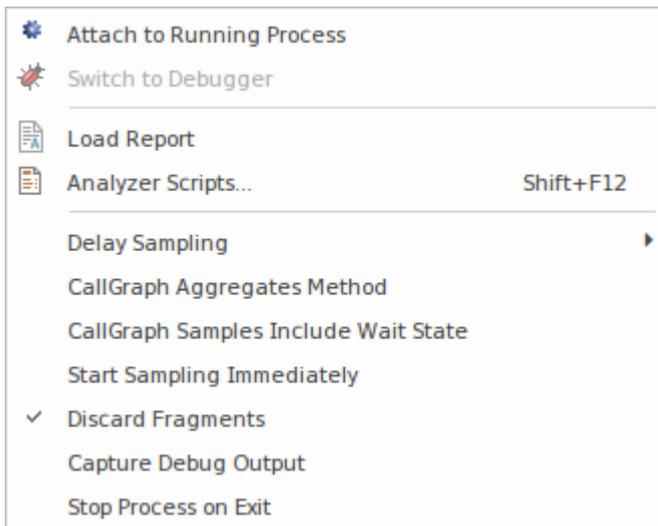
Options

Option	Description
Attach to Running Process	Select this option to display the 'Attach to Process' dialog, from which you choose an active process to Profile.
Switch to Debugger	Select this option to change operations from **Profiling** to Debugging. The **Debugger** has an equivalent drop-down menu option that you can use to switch from Debugging to Profiling.
Load Report	Select this option to load a previously saved report from the file system.
Analyzer Scripts	Select this option to open the **Analyzer Script window**, which is the model repository for configuring builds, debugging, and all other **Visual Execution Analyzer** options.
Delay Sampling	Select this option to set a delay between clicking on a 'Start Profiling' option and the **Profiling** actually beginning. The delay can be 3, 5 or 10 seconds. Select 'None' to cancel any delay set.
CallGraph Aggregates Method	When this option is selected, instances of the identical stack sequences are aggregated by method. That is to say, line numbers / instructions within a method are ignored, so two stacks will be counted as one where they differ only by line number in their final frame.
CallGraph Samples Include Wait State	When this option is selected, the **Profiler** will sample all threads, including those in Wait states. When unselected, the Profiler only samples threads that have accumulated CPU time since the last interval expired.

Start Sampling Immediately	Select this option to trigger Data Collection immediately on launch. You would typically use this option to profile a process during start up.
Discard Fragments	When stacks cannot be reconciled to the entry point of a thread they are referred to as fragments. The number of fragments encountered during sampling is displayed in the sampler **Summary window**. You can set this option to collect or discard fragments; when the Discard Fragments option is: • Selected, fragments do not appear in the reports, although the number encountered is still updated • Deselected, a special collection named 'fragments' is created in the call graph to house them, and to ensure they data is not mixed in with the complete samples
Capture Debug Output	(Applies to Process Sampling). When selected, output normally visible during debugging is captured and displayed in the **Debug window**. Note that only debug builds will typically emit debug output.
Stop Process on Exit	This option determines termination behavior for the **Profiler**. When the option is selected, the target process will terminate when the Profiler is stopped.

Start & Stop the Profiler

Profiling is a two stage process of data collection and reporting. In Enterprise Architect the data collection has the advantage of being a background task - so you are free to do other things while it runs. The information sent back to Enterprise Architect is stored until you generate a report. To view a report, the capture must be turned off. After the report is produced you can resume capture with the click of a button. If, for some reason, you decide to scrap your data and start again, you can do so easily and without having to stop and start the program again.
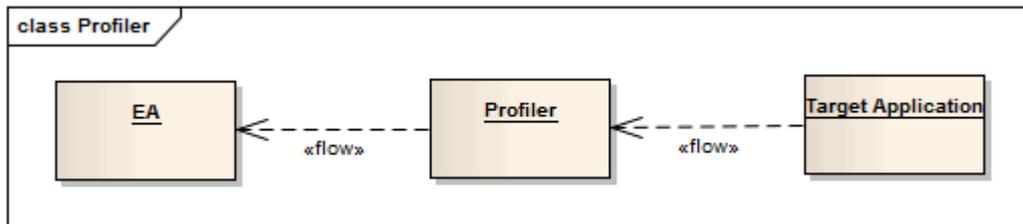
Access

Ribbon	Execute > Tools > **Profiler** > Open Profiler
Other	Execution Analyzer toolbar : Analyzer Windows | **Profiler**

Actions

Action	Detail
Toolbar	
Strategy Selection	Select the **Profiling** strategy from the available options on the Toolbar.
Start the Profiler	Click the **Run button** on the **Profiler window**
Stop the Profiler	The process exits if: • You click on the **Stop button** • The target application terminates, or • You close the current model If you stop the **Profiler** and the process is still running, you can quickly attach to it again.
Pause and Resume Capture	You can pause and resume capture at any time during a session. When capture is turned on, samples are collected from the target. When paused, the **Profiler** enters and remains in a wait state until either capture is enabled, the

	Profiler is stopped or the application finishes.
Generate Reports	The **Report button** is disabled during capture but is available when capture is turned off.
Clear Data Collection	You can clear any data samples collected and resume at any time. First suspend capture by clicking on the **Pause button**. The **Discard button**, as for the **Report button**, is enabled whenever capture is turned off. In clicking on the Discard button you will be asked to confirm the operation. This action cannot be undone.

Function Line Reports

After you have run the **Profiler** on an executing application and generated a Sampler report, you can further analyze the activity of a specific function listed in the report by generating a Function Line report from that item. A Function Line report shows the number of times each line of the function was executed. You produce one Function Line report at a time, on any method in the Sampler report that has a valid source file. The Function Line report is particularly useful for functions that perform loops containing conditional branching; the coverage can provide a picture of the most frequently and least frequently executed portions of code within a single method.

The line report you generate is saved when you save the Sampler report. The body of the function is also saved with the Function Line report to preserve the function state at that time.

Platforms supported

Java, Microsoft .NET and Microsoft native code

Create a Line Report

In the Sampler report, right-click on the name of the function to analyze, and select the 'Create Line report for function' option.

Once the **Profiler** binds the method, the Function Line report is opened on the **Sampler Report window**. The report shows the body of the function, including line numbers and text. As each line is executed a hit value will accumulate against that line. A timer will update the report approximately once every second.

| ⏮ ◀ ▶ ⏭ | Call Tree Report | Summary Report | ConsoleApplication::CQuickSort::Quicksort |

□ 💾 | ⊚

Module: ConsoleApplication
Function: CQuickSort::Quicksort
Date: 20/09/2013 2:53:21 PM
Author: smeagher
Iterations: 28679
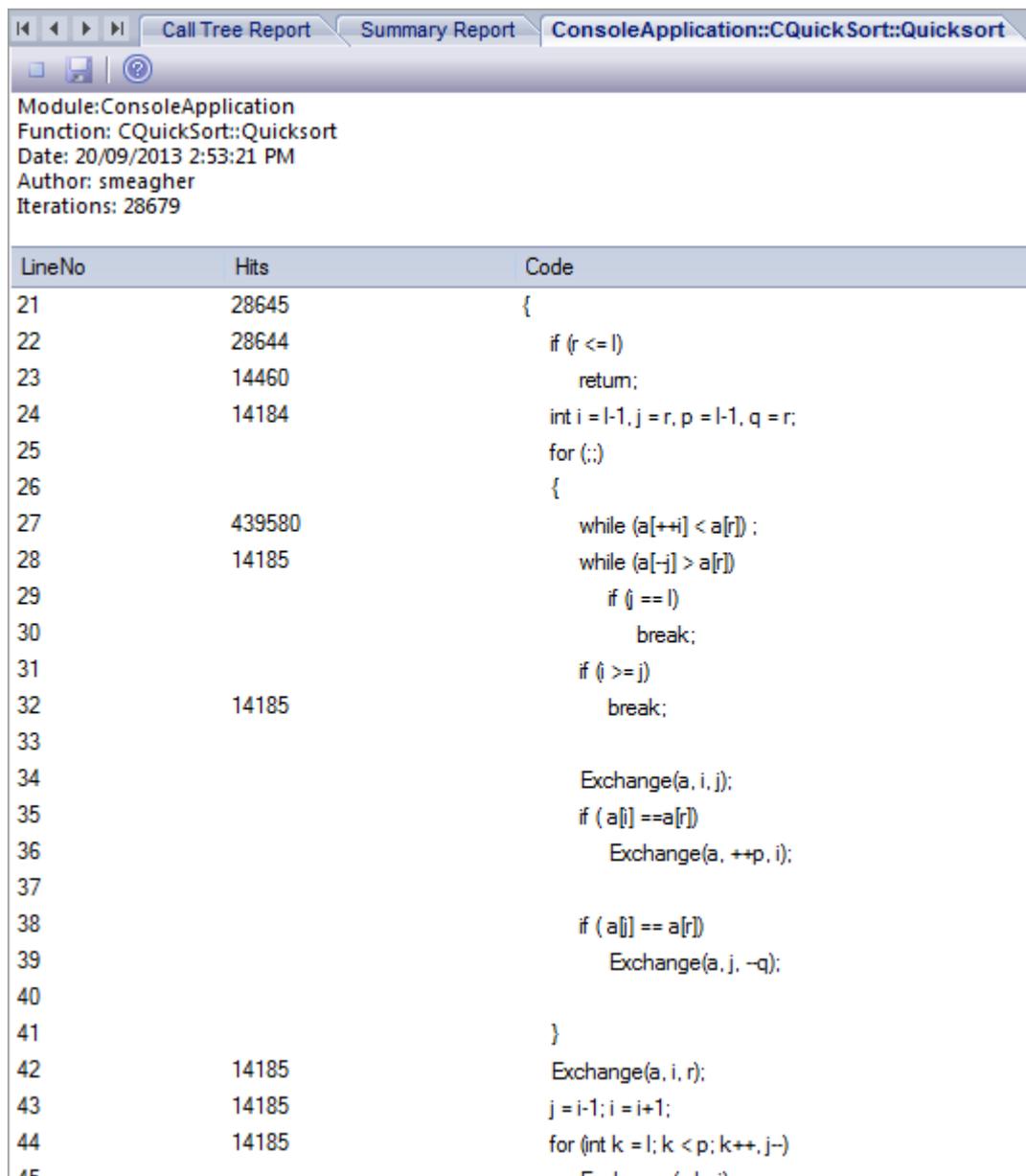
LineNo	Hits	Code
21	28645	{
22	28644	if (r <= l)
23	14460	return;
24	14184	int i = l-1, j = r, p = l-1, q = r;
25		for (;;)
26		{
27	439580	while (a[++i] < a[r]) ;
28	14185	while (a[--j] > a[r])
29		if (j == l)
30		break;
31		if (i >= j)
32	14185	break;
33		
34		Exchange(a, i, j);
35		if (a[i] ==a[r])
36		Exchange(a, ++p, i);
37		
38		if (a[j] == a[r])
39		Exchange(a, j, --q);
40		
41		}
42	14185	Exchange(a, i, r);
43	14185	j = i-1; i = i+1;
44	14185	for (int k = l; k < p; k++, j--)
45		

End Line Report Capture

Once enough information is captured, or the function has ended, click on the **Profiler** toolbar **Stop button** to stop recording the capture.

Save Reports

Use the **Save button** on the **Call Stack** toolbar to save the Sampler report and any Function Line reports to a file.

Delete Line Reports

Closing the 'Line Report' tab will close that report but the report data will only be deleted when the report is saved.

Generate, Save and Load Profile Reports

Reports can be produced at any time during a session, or naturally when a program ends. To enable the **Report button** while the program is running, however, you need to suspend **Profiling** by toggling the **Pause/Resume button**, or by terminating the **Profiler** with the **Stop button**. You have some options for reviewing and sharing the results:

- View the report
- Save the report to File
- Distribute the report as a **Team Library** resource
- Attach the report as a document to an Artifact element
- Synchronize the model by reverse engineering the source code that participated in the profile

Access

Ribbon	Execute > Tools > **Profiler** > Create Report from Current Data
Profiler	From the **Profiler window**, click on the ![icon] icon in the toolbar.

Load Report from File

The option is available from the drop down menu of the **Profiler** Window

Generate Report

From the **Profiler window**, click on the ![icon] icon in the toolbar.

CallFrequency Report

Call Stack	Inclusive Hits	Hits
⊟ xercesc_3_1::SAX2XMLReaderImpl::parse	16051	
⊟ xercesc_3_1::XMLScanner::scanDocument	16051	
⊟ xercesc_3_1::IGXMLScanner::scanDocument	16051	
⊟ xercesc_3_1::IGXMLScanner::scanContent	16051	
⊟ xercesc_3_1::IGXMLScanner::scanStartTagNS	16051	
⊟ xercesc_3_1::IGXMLScanner::resolveSchemaGrammar	16051	
⊟ xercesc_3_1::SchemaValidator::preContentValidation	16049	
⊟ xercesc_3_1::ComplexTypeInfo::checkUniqueParticleAttribution	16049	
⊟ xercesc_3_1::ComplexTypeInfo::makeContentModel	16049	
⊟ xercesc_3_1::DFAContentModel::DFAContentModel	16047	
⊟ xercesc_3_1::DFAContentModel::buildDFA	15998	515
⊟ xercesc_3_1::CMStateSet::operator|=	8174	8093
memcpy	32	32
⊞ xercesc_3_1::CMStateSet::allocateChunk	27	1
__security_check_cookie	21	21
TrailUpVec	1	1
⊞ xercesc_3_1::CMStateSet::~CMStateSet	3573	4
⊞ xercesc_3_1::XMemory::operator delete	841	2
xerces-c_3_1D	4416	2
xercesc_3_1::CMStateSet::getBit	1036	1036
⊞ xercesc_3_1::DFAContentModel::buildSyntaxTree	528	3
⊞ xercesc_3_1::CMStateSet::CMStateSet	373	3
xercesc_3_1::CMStateSet::getBitCountInRange	285	285
⊞ xercesc_3_1::XMemory::operator new	211	2
⊞ xercesc_3_1::CMStateSet::zeroBits	154	
⊞ xercesc_3_1::CMStateSetEnumerator::nextElement	153	136
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,⟩	59	2
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,⟩	28	2
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,⟩	25	
⊞ xercesc_3_1::DFAContentModel::makeDefStateList	25	2

Function Summary

Name	Inclusive Hits	Occurrences
mainCRTStartup	7408	1
__tmainCRTStartup	7407	1
xercesc_3_1::XMLFormatter::handleUnEscapedChars	7351	10
xercesc_3_1::XMLFormatter::formatBuf	7351	10
xercesc_3_1::XMLFormatter::specialFormat	7351	10
SAX2PrintHandlers::writeChars	7350	10
xercesc_3_1::XMLScanner::scanDocument	7350	1
main	7350	1
xercesc_3_1::SAX2XMLReaderImpl::parse	7350	1
xercesc_3_1::XMLScanner::scanDocument	7349	1
xercesc_3_1::IGXMLScanner::scanDocument	7348	1
xercesc_3_1::XMLFormatter::formatBuf	4042	8

Unfiltered Summary Report listing all participating functions in order of inclusive hits.

You can filter and reorganize the information in the report, in the same way as you do for the results of a **Model Search**.

Report Options

Right-click on the report to display the context menu.

Action	Detail
Show Source for Function	For the selected frame, select this option to display the corresponding line of code in a code editor. Frames that have source available are identifiable by their icon.
Find in Summary Window	Select this option to locate the function in the **Summary window**.
Collapse Graph	Select this option to collapse the entire graph including child nodes, visible or not.
Collapse to Node	Select this option to collapse the entire graph, then expand and set the focus to the

	selected node.
Follow Max Allocations	Select this option to expand an entire line in the graph.
Create Line Report for Function	Select this option to launch the **Profiler** (if it is not already running), immediately bind the selected function and ready it for recording. Once bound, an extra tab is opened in the current **Report View**. This report will update instantaneously, showing the number of times each line executed. Of course, the report will continue to record activity in the function even if is not visible. Note: In windowed programs, it is usually necessary to take some action in the application to cause the function to be invoked.
Create Function Graph	Select this option to create an additional tab, which shows the selected function in isolation. For a Call Frequency Profile, this produces a graph showing all the lines that led to this function being called (that is, the callers). For a Memory Profile, this produces a graph showing all lines that emanate from this function (that is, the callees).
Mark Initial Frame for Call Stack Diagram	Use prior to creating a **Call Stack** sequence diagram to limit the stack length. When this option is selected, the frame is marked and its text is highlighted. Frames above this one will then be excluded from any Sequence diagram produced.
Remove Mark	Removes the mark from a frame that was previously marked as 'Initial'.
Create Call Stack Diagram	Generates a sequence diagram for a single stack in the graph. The selected frame is depicted as the terminal frame in the stack. The initial frame of the stack defaults to the root node if no 'Initial' frame has been marked.
Create Weighted Call Graph Diagram	Generates a sequence diagram that presents a sequence for each visible stack branching from the selected frame. By expanding and collapsing the nodes of interest, a user can tailor the sequence diagram content to their liking.
Display the Heaviest Weighted Use	Select this option to display the line in the graph with the highest weight in which this function appears.
Display the Next Weighted Use	Select this option to navigate to the next line in the graph where the function appears. You can use the shortcut key combination **Ctrl+Down Arrow**.
Display the Previous Weighted Use	Select this option to navigate to the previous line in the graph where this function appears. You can also use the shortcut key combination **Ctrl+Up Arrow**
Save Report to File	Select this option to display the 'Save As' dialog, allowing you to choose where to store the report.
Save Report to Artifact	Note: Before selecting this option, go to the **Browser window** and select the Package or element under which to create the Artifact element. You are prompted to provide a name for the report (and element); type this in and click on the **OK button**. The Artifact element is created in the Browser window, underneath the selected Package or element. If you add the Artifact to a diagram as a simple link, when you double-click on the element the report is re-opened.

Team Library Options

Option	Description
Make Report a Team Library Resource	You can save any current report as a resource for a Category, Topic or Post in the **Team Library** to share and review at any time, as it is saved with the model. The report can also be compared with future runs. To begin this process, on the 'Team Library context menu select the menu option 'Share Resource | Add Active **Profiler** Report'.

Notes

- If you add the **Profiler** report to an Artifact element and also attach a Linked Document, the Profiler report takes precedence and is displayed when you double-click on the element; you can display the Linked Document using the 'Edit Linked Document' context menu option

Save Report in Team Library

You can save any current report as a resource for a Category, Topic or Document in the **Team Library**. The report can then be shared and reviewed at any time as it is saved with the model. This helps you to:

- Preserve a **Profiler** report to compare against future runs
- Allow other people to investigate the profile

Access

Context Menu	Right-click in **Team Library window** | Share Resource | Active **Profiler** Report