Enterprise Architect

User Guide Series

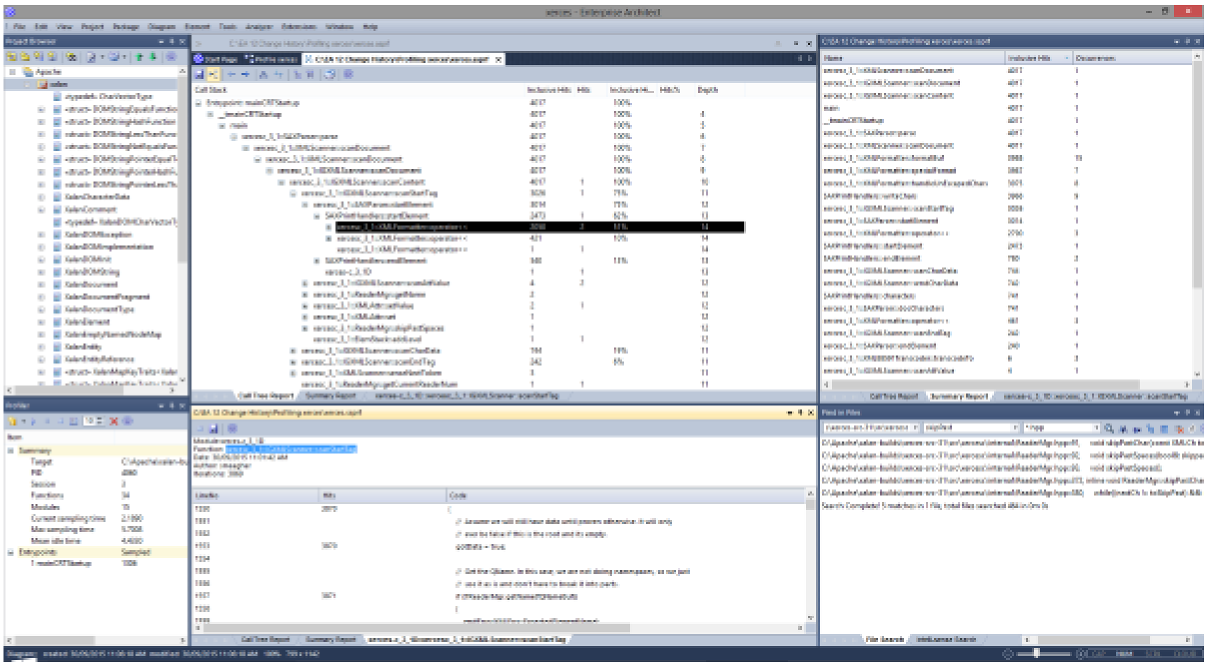
Profiling

Author: Sparx Systems
Date: 15/07/2016
Version: 1.0

CREATED WITH ENTERPRISE ARCHITECT

Table of Contents

Profiling



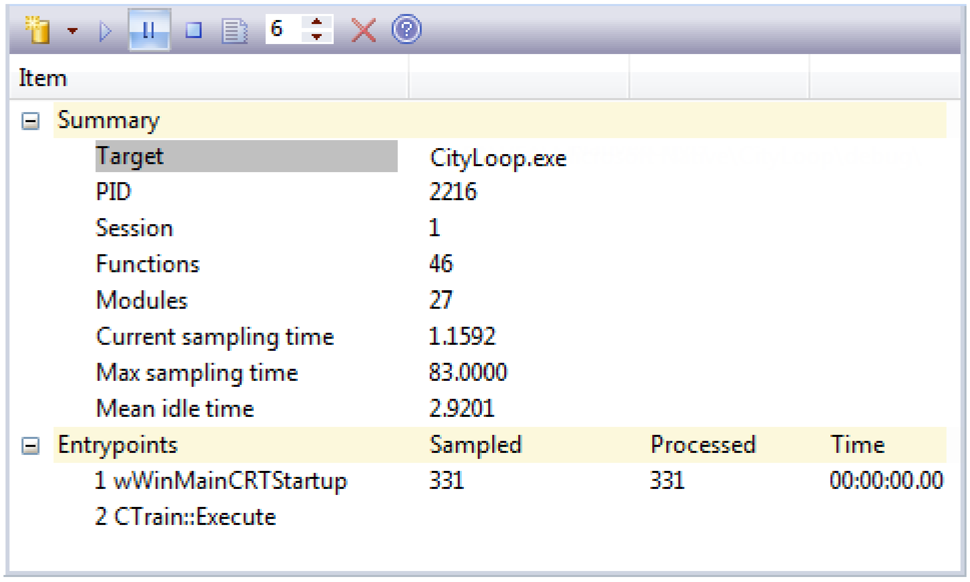
During the lifetime of software applications, it is not uncommon to investigate application tasks that are determined to be performing slower than expected. You might also simply want to know what is going on when you '*press this button*'! You can work this out quite quickly in Enterprise Architect 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, **Profiling** is the process by which the stacks of threads are sampled at regular intervals over a period of time to produce a data collection. The data is then analyzed to produce a weighted call graph. Behaviors are usually identifiable as root nodes (entrypoints) in the graph, or branches near these points. When viewed the report can be stored as a file, in either a binary or an XML format. They can also be stored within the model as Artifact elements, and as **Team Review** posts.

This image is an excerpt of a program that was profiled from startup. Although there are four concurrent threads running (not shown here), it is plain from the report that, other than the startup task, one or all threads are involved in a single behavior CTrain::Execute.

Access

Ribbon	Code > Analyzer > Profile Execute > Analyze > **Profiler** > Open Profiler
Menu	Analyzer | **Profiler**
Other	Execution Analyzer toolbar : Analyzer Windows | **Profiler**

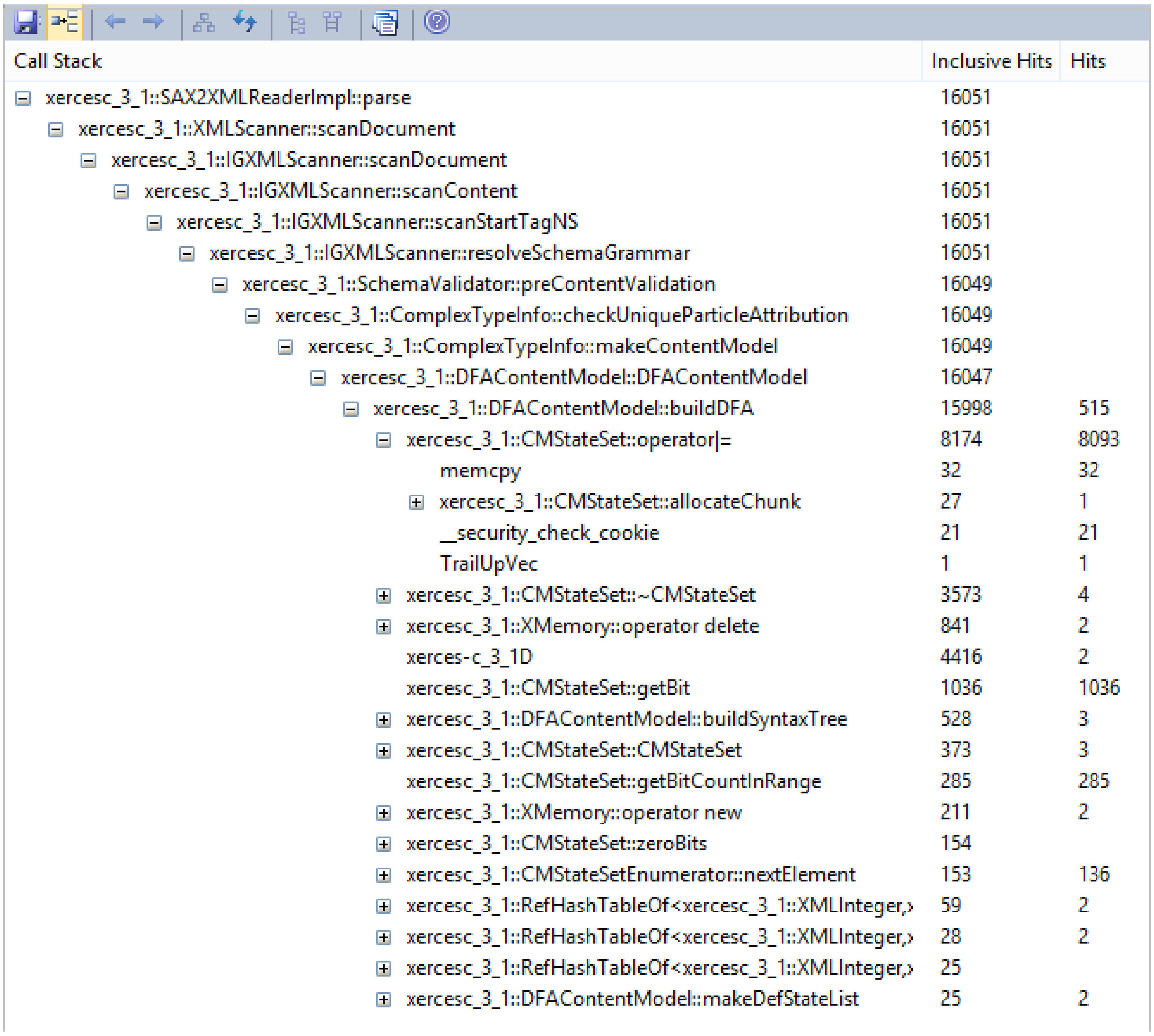
Capture



Item			
⊟ Summary			
Target	CityLoop.exe		
PID	2216		
Session	1		
Functions	46		
Modules	27		
Current sampling time	1.1592		
Max sampling time	83.0000		
Mean idle time	2.9201		
⊟ Entrypoints	Sampled	Processed	Time
1 wWinMainCRTStartup	331	331	00:00:00.00
2 CTrain::Execute			

The **Profiler** is controlled using its toolbar buttons. Here you can attach the Profiler to an existing process (or JVM), or launch the process for the active Analyzer Script. The Profiler window displays the details of the target process as it is profiled. These details provide some feedback, letting you see the samples which are (or are not) being taken. You also have a number of toolbar options for pausing and resuming capture, clearing captured data, and generating reports. You can get access to the reporting feature by pausing the capture. The reporting feature is disabled whilst data capture is in progress.

Weighted Call Graph



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

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. The weight of each branch 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 period captured. To understand the hit count consider this example of a profile where these three call stacks were detected.

A.B.C

A.B.D

A.B.D.E

The call stack A (implied) will have a hit count of 3

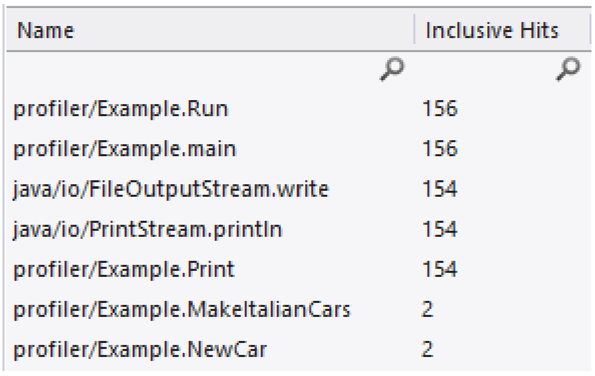
The call stack A.B (implied) will have a hit count of 3

The call stack A.B.D will have a hit count of 2

The call stack A.B.C will have a hit count of 1

The call stack A.B.D.E will have a hit count of 1

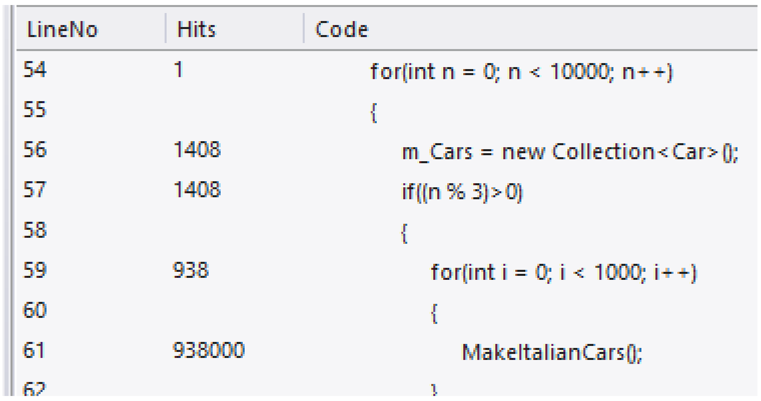
Function Summary Report



Name	Inclusive Hits
profiler/Example.Run	156
profiler/Example.main	156
java/io/FileOutputStream.write	154
java/io/PrintStream.println	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 presenting 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.

Notes

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

Learning Center topics

- **Alt+F1** | Enterprise Architect | Execution Analysis | **Profiling** Native Code | Introducing the **Profiler**

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 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 needs to have been launched with the Enterprise Architect profiling agent

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

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

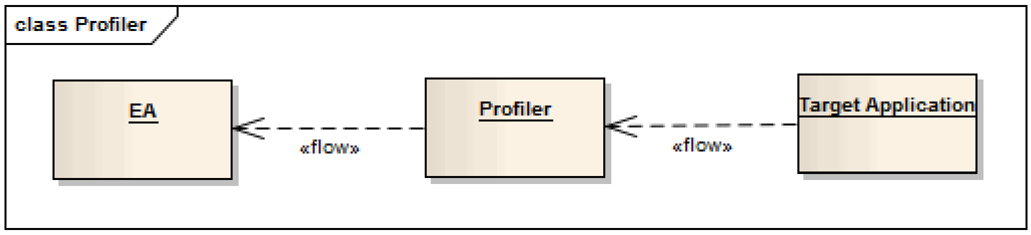
(Refer to the JDK documentation for details of the -agentpath VM startup option.)

Learning Center topics

- **Alt+F1** | Enterprise Architect | Execution Analysis | **Profiling** Native Code

Profiler Operation

Profiling is most usually controlled through the **Profiler** toolbar but similar options can be found easily in the Profile drop-down menu, on the Execute ribbon. 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 at the same time monitoring the profile and seeing how many samples have been captured. 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 wish to scrap your data and start again, you can do so easily and without having to stop and start the program again.



Access

Ribbon	Code > Analyzer > Profile Execute > Analyze > **Profiler** > Open Profiler
Menu	Analyzer | **Profiler**
Other	Execution Analyzer toolbar : Analyzer Windows | **Profiler**

Actions

Action	Detail
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 sample collection at any time during a session. When capture is turned on, samples are collected from the target. When paused, the profiler enters and remains idle. No samples are taken and nothing is transmitted. In short, the target is not interrupted.

Generate Reports	The report button is enabled whenever samples are collected and 1) capture is paused, 2) the **Profiler** process is stopped or 3) the Application ends.
Clear Data Collection	You can clear any data samples collected and resume at any time. First disable suspend capture by clicking the pause button. The **Discard button**, like the **Report button** is enabled whenever capture is turned off. In clicking the discard button you will be asked to confirm the operation. This action can not be undone.

Getting Started

When you run a **Profiling** session, almost every option you might need is available from the **Profiler** window toolbar. You can, for example, initiate the profiling session, attach to an already-running process, pause and resume profiling, stop the session, generate and view the Profile report or load a previously-generated report. You can also set Profiler options to modify the operation of the Profiler.

Access

Ribbon	Code > Analyzer > Profile Execute > Analyze > **Profiler** > Open Profiler
Menu	Analyzer | **Profiler**
Other	Execution Analyzer toolbar : Analyzer Windows | **Profiler**

Toolbar Buttons

Button	Action
▤ ▾	Set Profiler options, using a drop-down menu; the options are: • 'Attach to Running Process' - attach to and profile a process that is already running • 'Switch to debugger' - (enabled when you are running the **Profiler**) end the profiling session and attach the debugger to the running process; available on Microsoft Native and Microsoft .NET platforms • 'Load Report' - load and display a previously-generated report from an XML disk file • 'Analyzer Scripts' (**Shift+F12**) - display the **Execution Analyzer window** to create or edit scripts and configure the debugger • 'Start Sampling Immediately' - begin sample collection immediately upon either process start (main thread entry point executed) or attachment of process by the Profiler • 'Capture Debug Output' - capture any appropriate debug output and redirect it to the **System Output** window • 'Stop Process on Exit' - select to terminate the target process when the Profiler is stopped
▷	(When an application is configured for the Package) create the **Profiler** process, which launches the configured application.
⏸ ⏸	When the application is running, pause and resume sample capture. Pausing sampling activates the Report and Discard Data buttons.
◼	Stop the **Profiler** process; if any samples have been collected, the **Report button** is enabled.

📄	Generate a report on the current number of samples collected.
5 ⬍	Set the interval, in milliseconds, at which samples are taken of the target process; the range of possible values is 1 - 250.
✕	Discard the collected data. You are prompted to confirm the discard.
❓	Display the Help topic for this window.

Learning Center topics

- **Alt+F1** | Enterprise Architect | Execution Analysis | **Profiling** Native Code | Profile Application Startup
- Alt+F1 | Enterprise Architect | Execution Analysis | Profiling Native Code | Profile Running Application
- Alt+F1 | Enterprise Architect | Execution Analysis | Profiling Native Code | Introducing the **Profiler** | Set Capture Options

Generate, Save and Load Profile Reports

Once you have collected some samples you can prepare a report at any time. To enable the reporting button, you need to suspend active sampling. You can do this 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.

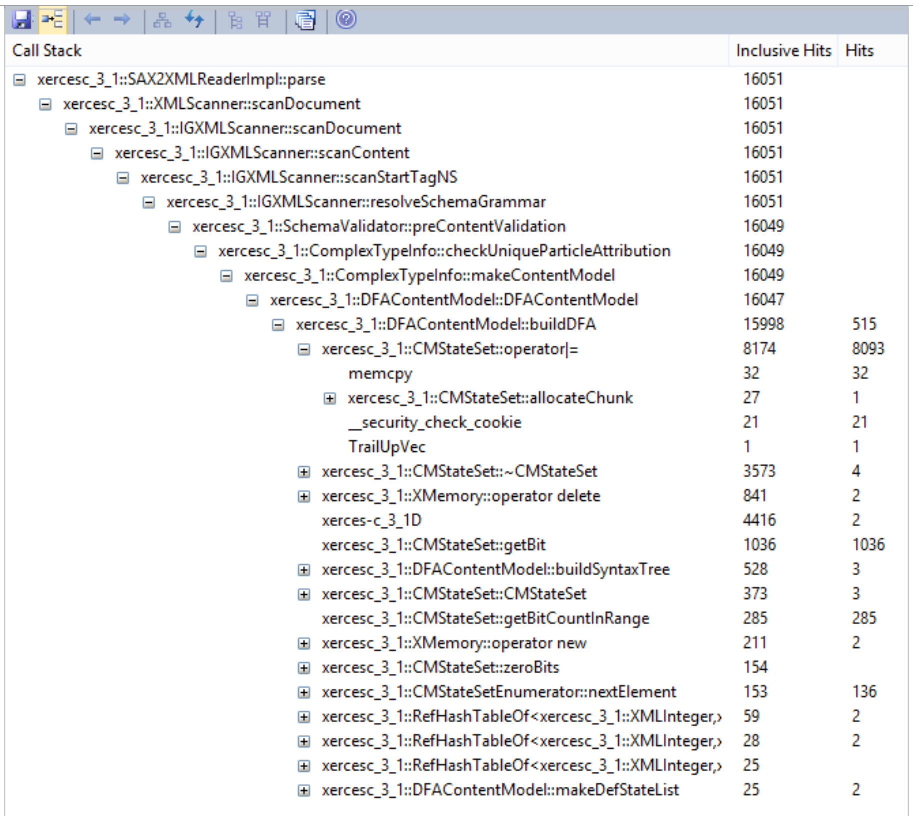
- View the report
- File the report in binary or XML format
- Distribute the report as a **Team Review** resource
- Attach the report as a document to an Artifact element
- Synchronize the model by reverse engineering the source code that participated in the profile.

Access

Ribbon	Execute > Analyze > **Profiler** > Create Report from Current Data
Menu	Analyzer | Profile | Display **Profiler** Report
Other	From the '**Profiler**' window, click on the icon in the toolbar.

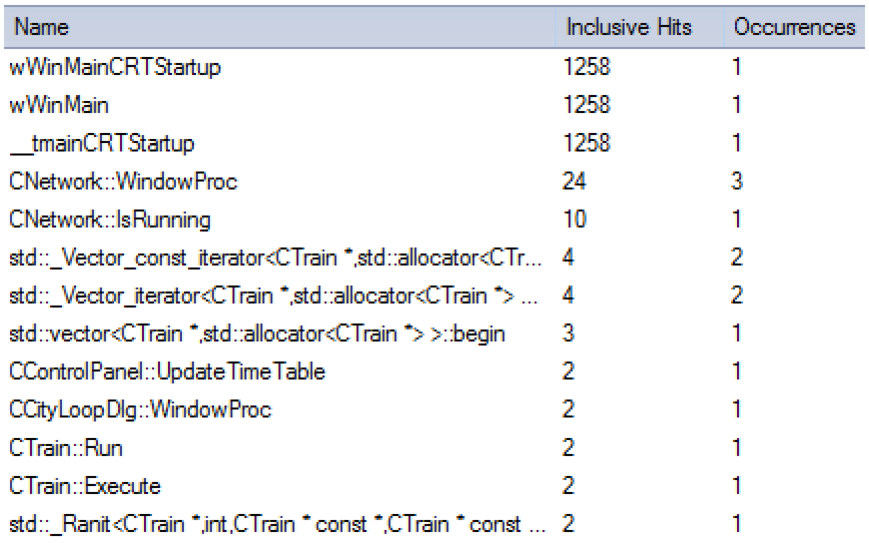
Options

Action	Detail
Display Report	Click on the icon in the **Profiler** toolbar. The generated report displays two views; a weighted call graph



```
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
```

and a summary view:



Name	Inclusive Hits	Occurences
wWinMainCRTStartup	1258	1
wWinMain	1258	1
__tmainCRTStartup	1258	1
CNetwork::WindowProc	24	3
CNetwork::IsRunning	10	1
std::_Vector_const_iterator<CTrain *,std::allocator<CTr...	4	2
std::_Vector_iterator<CTrain *,std::allocator<CTrain *> ...	4	2
std::vector<CTrain *,std::allocator<CTrain *> >::begin	3	1
CControlPanel::UpdateTimeTable	2	1
CCityLoopDlg::WindowProc	2	1
CTrain::Run	2	1
CTrain::Execute	2	1
std::_Ranit<CTrain *,int,CTrain * const *,CTrain * const ...	2	1

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

Show or hide unknown frames	By default the report excludes unknown function calls and frames in the call tree. Click on the toolbar button to show or hide these frames.
Display the previous or next incidence of a function	When you select a function, and a call to that function is present in more than one stack, you can click on the and buttons to navigate between all stacks in which the function appears. Navigation is from highest use to lowest.

Set a Node as the root	If you want to examine a deeply-nested branch, you can make it the root node of the report. To do this, click on the node and click on the button. Only function calls emanating from the selected branch will be displayed.
Reset the Root Node	To reset the report to normal click on the button.
Generate a Sequence Diagram	You can generate a Sequence diagram from any node in the **Profiler** report. To do this, either: • Right-click on the node and select the 'Create **Sequence Diagram**' option • Click on the node and click the button in the toolbar The generated Sequence diagram reflects all activity resulting from the selected node. It is created as a child diagram of the Interaction corresponding to the node, and is displayed immediately.
Save Report to File	Either: • Click on the button • Select the 'Save Report to File' context menu option The 'Save As' dialog displays. Type the file name of the report and choose whether to export as a binary or a text file. If you use the binary format the file is smaller, but can only be viewed through an edition of Enterprise Architect. If you choose the XML format the file might be very large, but of course can be viewed in many text editors.
Load a Saved Report	Click on the button or the 'Load Report' context menu option. The 'Open' dialog displays, on which you browse for and select the report file. Click on the **Open button**; the **Call Stack** opens or refreshes with the loaded report.
Generate a Function 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 line report is opened as an additional tab to the current report View. Function Line reports are saved together with the sample data and will be displayed whenever the file or element is reloaded.
Make Report a Team Review Resource	You can save any current report as a resource for a Category, Topic or Post in the **Team Review** 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, select the menu option 'Team Review Context Menu | Share Resource | Add Active **Profiler** Report'.
Attach Report to an Artifact Element	In the **Project Browser**, select the Package or element under which to create the Artifact element. On the **Call Stack** window showing the report, right-click and select the 'Save Report to Artifact' option. 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 Project Browser, 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 Call Stack window displays, showing the saved report.
Synchronize Code with Model	During its operation, the **Profiler** generates a collection of relevant code files, which you can reverse-engineer to the current model in a single operation using the 'Synchronize Model' dialog. Click the button on the toolbar to use this feature.

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

Learning Center topics

- **Alt+F1** | Enterprise Architect | Execution Analysis | **Profiling** Native Code | View Report
- Alt+F1 | Enterprise Architect | Execution Analysis | Profiling Native Code | Load Report from Disk

Setting Options

Topics

Topic	Icon
Interval	5 Detail: Set the interval, in milliseconds, at which samples are taken of the target process; the range of possible values is 1 - 250.
Profile Options	Detail: Set Profiler options, using a drop-down menu; the options include: • 'Start Sampling Immediately' - begin sample collection immediately upon either process start (main thread entry point executed) or attachment of process by **Profiler** • 'Capture Debug output' - capture any appropriate debug output and redirect it to the **System Output** window • 'Stop Process on Exit' - select to terminate the target process when the Profiler is stopped

Learning Center topics

- **Alt+F1** | Enterprise Architect | Execution Analysis | **Profiling** Native Code | Introducing the **Profiler** | Set Capture Options

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 Sampler report by generating a function line report from that report 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 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 line report to preserve the function state at that time.

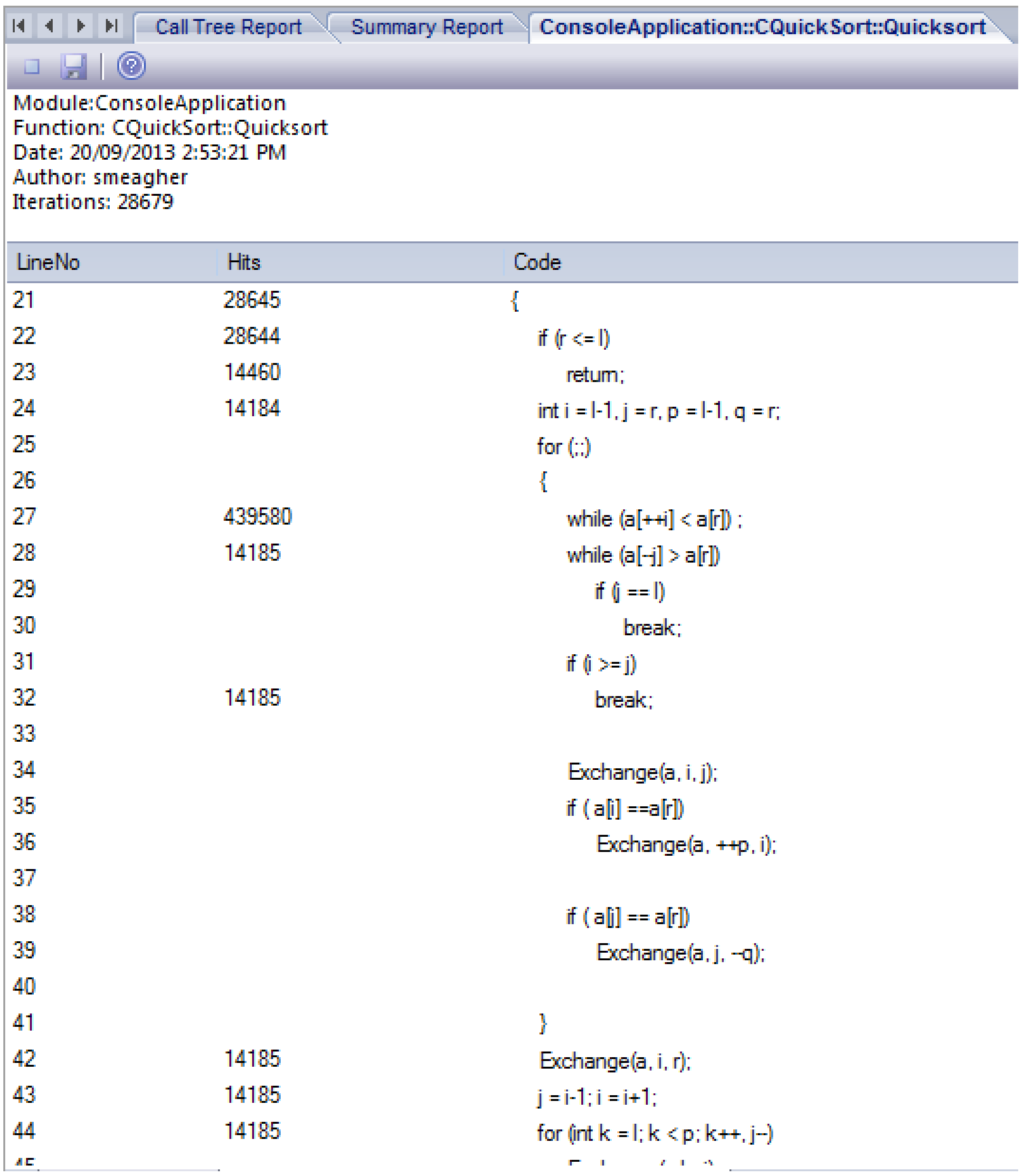
Platforms supported

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 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.



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--)

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.

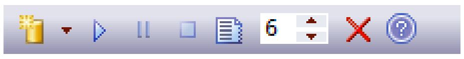
Delete Line Reports

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

Start & Stop the Profiler

For most debugging operations it is necessary to have first configured an Execution Analyzer Script that typically defines the application to build, test and debug, and any sequence recording options. It is possible to use the **Profiler** without doing any of this by using the Attach to Process option in the drop-down option list.

If the application to profile is the one defined in the current Package, use the Run Profiler button.



- - (When an application is configured for the Package) create the Profiler process, which launches the configured application
- - Stop the Profiler process

Learning Center topics

- **Alt+F1** | Enterprise Architect | Execution Analysis | **Profiling** Native Code | Profile Running Application

Save Report in Team Review

You can save any current report as a resource for a Category, Topic or Document in the **Team Review**. 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 Review** window | Share Resource | Active **Profiler** Report