Configuring Precise for Microsoft .NET

This section includes the following topics:

About configuring Precise for Microsoft .NET

When we discuss configuring Precise for Microsoft .NET, we talk about two sections:

- Dynamic instrumentation
- Tracking instrumentation activity

The Microsoft .NET AppTier Collectors agents use the dynamic instrumentation of your Microsoft .NET-based applications to track software activities that are based on Microsoft .NET. Therefore, the instrumentation configuration determines the tracking quality, the amount of overhead involved, and whether the data collected really suits your specific needs.

About the dynamic configuration files

The rules for dynamic instrumentation configuration are defined in the following files:

- **instrumentation.xml**. This file contains explicit instrumentation rules for the tracked instances
- **default-instr-config.xml**. This file contains the default instrumentation rules for the tracked instances.

  > Do not modify this file without consulting Precise Technical Support.

- **ActivityCollector.xml**. This file contains the configuration of the Microsoft .NET AppTier Collector agent, which determines other tracking rules, including implicit instrumentation rules.

Limitations of tracking instrumentation activity

Tracking instrumentation activity is subject to the following limitations:

- Only the Microsoft .NET version 1.1, 2.0, 3.0, 3.5, 4.0, and 4.5 Frameworks are supported.
- The time resolution (granularity) of the tracked activity is approximately 16 milliseconds.
- Certain system DLLs that are part of the Microsoft .NET infrastructure cannot be instrumented. It is strongly recommended not to add an additional system DLL to the DLL list without first contacting Precise Technical Support.

  > All system DLLs that are discussed, are safe for instrumentation.

- Generated Microsoft .NET wrappers for COM+ components that are produced by Microsoft .NET Framework SDK utilities (such as TLBImp) cannot be tracked.
- "Pre-Jitted" assemblies cannot be instrumented because they were compiled beforehand. For example, the System.Windows.Forms.dll is pre-jitted in the Microsoft .NET framework.

About the instrumentation file

The Instrumentation.xml file is located in the `<i3_root>\products\dotnet\config` directory. It contains the Instance-specific section.

Following is an example of the file structure:

```
<?xml version="1.0" encoding="utf-8"?>
<instrumentation-config>
<instances>
    <instance name="Instance_A">
        Instance A instrumentation rules
    </instance>
    <instance name="Instance_B">
        Instance B instrumentation rules
    </instance>
</instances>
</instrumentation-config>
```

The file consists of the following subsections:

- **DLL list**. A list of DLLs to be instrumented and specific configuration rules for each DLL. If the underlying process of the instance should be instrumented, it must be mentioned explicitly.
- **Common instrumentation rules (“instrument”)** for the whole DLL list.
- **Common exclude rules (“ignore”)**. Following is an example of the file structure:

```
<dlls>
    <dll name="dll_A_name" />
    <dll name="dll_B_name" />
    <dll name="dll_C_name" >
        <instrument>
            Specific instrumentation rules for DLL C.
        </instrument>
        <ignore>
            Specific ignore rules for DLL C.
        </ignore>
    </dll>
    <dll name="dll_D_name" >
        <instrument>
            Specific instrumentation rules for DLL D.
        </instrument>
        <ignore>
            Specific ignore rules for DLL D.
        </ignore>
    </dll>
</dlls>
<instrument>
            Common instrumentation rules for all the DLLs in this context (A, B, C, D).
</instrument>
<ignore>
            Common ignore rules for all the DLLs in this context (A, B, C, D).
</ignore>
```

Example of the instrumentation.xml file

In the following example, all DLLs with the prefix “Pet” are instrumented and tracked, regardless of their path. Likewise, for the classes with the prefix “Pet”, all methods are instrumented and tracked.

```
<?xml version="1.0" encoding="utf-8"?>
<instrumentation-config>
    <instances>
        <instance name="AspNetIIS5">
            <dlls>
                <dll name="Pet*" />
            </dlls>
            <instrument>
                <classes>
                    <class>
                        <name>Pet*</name>
                        <called-method>
                            <methods>
                                <method>
                                    <name>*</name>
                                </method>
                            </methods>
                        </called-method>
                    </class>
                </classes>
            </instrument>
        </instance>
    </instances>
</instrumentation-config>
```

About the instrumentation.xml file tags

The following table describes the tags in the Instrumentation.xml file.

Table 13-1 Instrumentation file tags

Tag name	Description	Comment
instrumentation-config	The top level XML tag.	
instances	The list of instances to be instrumented.	

`instance name` (attribute)	The logical name of the instance.	
`instance tracker` (attribute)	The tracker DLL's reference to use.	Optional tag.
dlls	The list of DLLs to be instrumented when loaded by this instance.	Without a list of DLLs, all loaded DLLs should be instrumented.
`dll name` (attribute)	The name of the DLL.	Pattern match (for example `Company.GUI.*`) If the main module of the process must also be instrumented, the process name should be added explicitly to this DLL list.
`dll path` (attribute)	The path of the DLL.	Pattern match. Optional tag. Without this tag, all DLLs with the indicated name are instrumented.
`dll version` (attribute)	The version of the DLL.	Pattern match. Optional tag. Without this tag, all DLLs with the indicated name are instrumented.
instrument	The include list for instrumentation.	
interfaces	List of interfaces to be instrumented.	All classes that implement these interfaces must be instrumented.
interface	A specific interface (item in the interfaces list).	
classes	The list of classes to be instrumented. Also classes that inherit from these classes must be instrumented.	The same syntax as the interfaces section.
class	A specific class (item in the classes list).	
`name` (in interface /class section)	The specific name of the interface/class.	Pattern match. (for example, using * as the name indicates all names in this context)
caller-method	The list of methods to be instrumented on the callee side.	
methods	The beginning of the list.	
method	The description of a specific method (within the classes/interfaces context).	
type	The type of method invocation.	Optional tag (the default is Custom). Specified in the `default-inst-config.xml` file.
`name` (of method)	The specific name of the method.	Pattern match (for example, using * as the method name indicates all methods in this context).
params	The list of parameter types.	
param	A specific parameter type.	
all-calls-to-method	The list of methods to be instrumented on the caller side.	The same syntax as the caller-method section.
ignore	The exclude list for instrumentation.	The syntax of this section and its subcomponents is identical to the instrument section.

About instrumenting DLLs from the GAC

The Global Assembly Cache (GAC) is a logical directory that stores all assemblies that can be shared among applications. An assembly is placed in the GAC at deployment time, using either an installer that knows about the assembly cache or the Global Assembly Cache Utility (gacutil.exe), found in the MS .NET Framework SDK.

To instrument a DLL from the GAC, do not specify the path attribute in the DLL list. Instead, use only the DLL name. To instrument a DLL with a specific version, use the version attribute.

About instrumenting DLLs of an ASP.NET application

The ASP.NET DLLs are typically loaded from a special temporary directory, such as:

```
C:\WINNT\Microsoft.NET\Framework\v1.1.4322\Temporary ASP.NET
Files\mspetshop\c73f3fa0\1ae4c2f3\assembly\dl2\bee75c45\aa2e9da0_9807c501)
```

As in the case of the GAC, to instrument such DLLS, do not specify the ASP.NET DLL path and only use the DLL name.

About instrumenting standalone applications (Console, WinForm, Windows services)

A standalone application that starts with a main method usually does not finish before the application terminates. This main method typically invokes other methods that actually perform most of the activities in the application. These other methods are started and stopped quite frequently, but their top-level caller (the main method) remains running.

The Microsoft .NET AppTier Collector agent is designed to report only when a complete sequence of method invocations has finished. To deal with this kind of scenario, the `ignore` tag is used in the default instrumentation file to exclude the instrumentation and the tracking of the main method.

About callee vs. caller-side instrumentation (all-calls-to-method)

Callee-side instrumentation means instrumenting the methods (their prologues and epilogues) to track the method invocations. Caller-side instrumentation means instrumenting the calls to the methods to track the called method invocations. These techniques are two different ways to gather the same data.

Precise for Microsoft .NET generally uses the callee-side instrumentation technique. If callee-side instrumentation is not possible, caller-side instrumentation can be applied instead. To do so, replace the default called-methods tag with the all-calls-to-method tag. This alternative is best used when the tracked method is part of the Microsoft .NET framework system DLLs that cannot be instrumented directly, such as the System.dll.

About tracking service requests (URLs)

Service Requests are tracked by default. The ability to track Service Requests (URLs) relies on the following blocks, which are part of the default instrumentation file:

```
<dll name="System.Web.dll" version="1.*">
    <instrument>
        <classes>
            <class>
                <name>System.Web.HttpRuntime</name>
                    <called-method>
                        <methods>
                            <method type="HTTP">
                                <name>ProcessRequest</name>
                            </method>
                        </methods>
                    </called-method>
            </class>
        </classes>
    </instrument>
</dll>
<dll name="System.Web.dll" version="2.*">
    <instrument>
        <classes>
            <class>
                <name>System.Web.HttpRuntime</name>
                    <called-method>
                        <methods>
                            <method type="HTTP">
                                <name>ProcessRequestNoDemand</name>
                            </method>
                        </methods>
                    </called-method>
            </class>
        </classes>
    </instrument>
</dll>
```

The instrumentation configuration partially filters the tracking of Service Requests: Service Requests are only tracked if the respective Web application's DLLs are listed in the instrumentation file.

About tracking SQL statements

SQL statements are tracked by default. The ability to track SQL statements relies on the following blocks, which are part of the default instrumentation file:

```
<dlls>
<dll name="System.Data.Oracleclient.dll">
    <instrument>
        <classes>
            <class>
    <name>System.Data.OracleClient.OracleCommand</name>
                <called-method>
                    <methods>
                        <method type="ADO">
    <name>ExecuteOracleNonQuery</name>
                        </method>
```

```
                    <method type="ADO">
                        <name>ExecuteOracleScalar</name>
                    </method>
                    <method type="ADO">
                        <name>ExecuteReader</name>
                    </method>
                    <method type="ADO">
                        <name>ExecuteScalar</name>
                    </method>
                    <method type="ADO">
                        <name>ExecuteNonQuery</name>
                    </method>
                    <method type="ADO">
                        <name>Prepare</name>
                    </method>
                </methods>
            </called-method>
        </class>
    </classes>
</instrument>
</dll>
<dll name="System.Data.dll">
    <instrument>
        <classes>
            <class>
                <name>System.Data.SqlClient.SqlCommand</name>
                <called-method>
                    <methods>
                        <method type="ADO">
                            <name>ExecuteReader</name>
                        </method>
                        <method type="ADO">
                            <name>ExecuteXmlReader</name>
                        </method>
                        <method type="ADO">
                            <name>InternalExecuteNonQuery</name>
                        </method>
                    </methods>
                </called-method>
            </class>
            <class>
                <name>System.Data.Common.DataAdapter</name>
            <called-method>
                    <methods>
                        <method type="ADO">
                            <name>Fill</name>
                        </method>
                        <method>
                            <name>Update</name>
                        </method>
                    </methods>
                </called-method>
            </class>
        </classes>
    </instrument>
</dll>
<dll name="Oracle.DataAccess.dll">
    <instrument>
        <classes>
            <class>
                <name>Oracle.DataAccess.Client.OracleCommand</name>
                <called-method>
                    <methods>
                        <method type="ADO">
                            <name>ExecuteReader</name>
                        </method>
                        <method type="ADO">
                            <name>ExecuteScalar</name>
                        </method>
                        <method type="ADO">
                            <name>ExecuteNonQuery</name>
                        </method>
                        <method type="ADO">
                            <name>ExecuteXmlReader</name>
                        </method>
                    </methods>
                </called-method>
            </class>
        </classes>
```

```
    </instrument>
</dll>
</dlls>
<!-- Instrument all the methods of classes which implement the following interfaces. -->
<instrument>
    <interfaces>
    <!-- Uncomment the following section to measure ADO.NET Open Connection response times -->
        <!-- <interface>
            <name>System.Data.IDbConnection</name>
            <called-method>
                <methods>
                    <method type="ADO">
                        <name>Open</name>
                    </method>
                </methods>
            </called-method>
        </interface> -->
        <interface>
            <name>System.Data.IDbCommand</name>
            <called-method>
                <methods>
                    <method type="ADO">
                        <name>ExecuteReader</name>
                    </method>
                    <method type="ADO">
                        <name>ExecuteScalar</name>
                    </method>
                    <method type="ADO">
                        <name>ExecuteNonQuery</name>
                    </method>
                    <method type="ADO">
                        <name>Prepare</name>
                    </method>
                </methods>
            </called-method>
        </interface>
    </interfaces>
</instrument>
```

These blocks define the DLLs and methods that are used for SQL statement extractions. This configuration supports the following ADO.NET providers:

- Microsoft provider for SQL-Server connectivity
- Microsoft provider for Oracle connectivity
- Microsoft ODBC connectivity
- Microsoft OLEDB connectivity.
- Oracle provider for Oracle connectivity

All SQL statements the above ADO.NET providers and methods use are tracked, even if the methods that call these ADO.NET methods are not explicitly selected for tracking in the Instrumentation.xml file.

The default instrumentation file for Precise for Microsoft .NET does already contain these blocks. If your application uses other database providers, consult Precise Technical Support.

About tracking COM+ (Enterprise services)

COM+ components can be developed and used in the following way:

- The COM+ library components are executed and used as a library (DLL) in the context of an executable. The calls to the COM+ component methods look like other intra-process method calls.
- The COM+ server components are hosted in an external COM+ container. A call from the COM+ component client (process) to the hosted COM+ component (external server process) is an inter-process call. In other words, activity exists in two different processes: the client-side process (caller-side) and the server-side process (callee-side).

To have the Microsoft .NET infrastructure reuse COM+ technology, perform the following tasks:

1. Develop Microsoft .NET-based DLLs (Enterprise Services) on top of the COM+ engine.
2. Reuse COM+ components with Microsoft .NET interoperability mechanisms (this case equals other interoperability cases and are not discussed here).

Calls to COM+ library components that were developed in Microsoft .NET equal other Microsoft .NET DLLs (for instrumentation purposes). To track the calls to the methods of these components, you only need to add the appropriate DLLs to the <dlls> list of the instance and specify the classes and methods to be tracked in the <instrumentation> block.

To track COM+ server components that are hosted in the COM+ container, perform the following tasks:

1. To track server-side activity (the activation of the components' methods), add a new instance to the server for the COM+ container using Precise Agent Installer. The underlying process name of the COM+ host is dllhost.exe. In Precise Agent Installer, use this process name, but do not

specify the path.
The <dlls> list of this new instance must include all DLLs of the Microsoft .NET-based COM+ server components to be tracked. The <instrument> block must specify the classes and methods that need to be instrumented.

2. To track client-side activity (the calls to the COM+ components), add the DLLs of the appropriate COM+ component to the client instance (with the appropriate <instrument> block). In addition, insert the following section under the instrumentation section for the COM+ client instance:

```
<dll name="System.EnterpriseServices.dll">
    <instrument>
        <classes>
            <class>
                <name>System.EnterpriseServices.ServicedComponentProxyAttribute</name>
                    <called-method>
                        <methods>
                            <method>
                                <name>CreateInstance</name>
                            </method>
                        </methods>
                    </called-method>
            </class>
            <class>
                <name>System.EnterpriseServices. RemoteServicedComponentProxy</name>
                    <called-method>
                        <methods>
                            <method>
                                <name>Invoke</name>
                            </method>
                        </methods>
                    </called-method>
            </class>
        </classes>
    </instrument>
</dll>
```

About tracking Web services (calls)

The callee-side Web service infrastructure (Web service server-side) is implemented in ASP.NET. In other words, tracking Web services in the called-side is covered by tracking Service Requests. See About the instrumentation file.

The Microsoft .NET framework contains comprehensive support for Web service clients (caller-side). For example, Visual Studio.NET generates a proxy class code that wraps the calls to the underlying Web service (the generated class extends an appropriate base class, such as System.Web.Services. Protocols.SoapHttpClientProtocol).

Web services are tracked by default. The ability to track Web service calls for all instances relies on the following blocks, which are part of the default instrumentation file:

```
<dll name="System.Web.Services.dll" >
    <instrument>
        <classes>
            <class>
                <name>System.Web.Services.Protocols. SoapHttpClientProtocol</name>
                <called-method>
                <methods>
                    <method type="WS">
                        <name>Invoke</name>
                    </method>
                    <method type="WS">
                        <name>BeginInvoke</name>
                    </method>
                    <method type="WS">
                        <name>EndInvoke</name>
                    </method>
                </methods>
                </called-method>
            </class>
            <class>
                <name>System.Web.Services.Protocols. HttpSimpleClientProtocol</name>
                <called-method>
                <methods>
                    <method type="WS">
                        <name>Invoke</name>
                    </method>
                    <method type="WS">
                        <name>BeginInvoke</name>
                    </method>
                    <method type="WS">
                        <name>EndInvoke</name>
                    </method>
                </methods>
            </called-method>
            </class>
        </classes>
    </instrument>
</dll>
```

About tracking the message queue API

To have Precise for Microsoft .NET track the usage of the Microsoft Message Queue API, you need to instrument the System.Messaging.dll file by manually adding the following blocks to the instrumentation block of the specific instance in the Instrumentation.xml file. Usually, it is sufficient to only instrument and track the methods Send and Receive to ensure that all appropriate Microsoft Message Queue call are tracked.

The default instrumentation file for Precise for Microsoft .NET does not contain these blocks. To track Microsoft Message Queue calls, you must add these blocks manually.

To instrument Message Queue calls, insert the following section under instrumentation rules for the monitored instance in the instrumentation file:

```
<dll name="System.Messaging.dll">
    <instrument>
        <classes>
            <class>
                <name>System.Messaging.MessageQueue</name>
                    <called-method>
                        <methods>
                            <method>
                                <name>Send</name>
                            </method>
                            <method>
                                <name>Receive</name>
                            </method>
                            <method>
                                <name>BeginReceive</name>
                            </method>
                            <method>
                                <name>EndReceive</name>
                            </method>
                        </methods>
                    </called-method>
            </class>
        </classes>
    </instrument>
</dll>
```

About the ActivityCollector.xml file

The ActivityCollector.xml file is the main configuration file of the Microsoft .NET AppTier Collector agent that gathers activity information. It is composed of the following logical sections:

- **Aggregator settings**. Contains the settings that are grouped under the aggregator tag.
- **Tracker settings**. Contains the settings that are grouped under the tracker tag.

The Aggregator settings sections are read when the Collector agent is loaded. Therefore, if you modify these sections, you must restart the Collector agent through Precise Agent Installer or by restarting the Precise for Microsoft .NET Collector service using the Services window.

The Tracker settings are read when the tracked Microsoft .NET instance (that is, the underlying Microsoft .NET-based process) is loaded. Therefore, if this section is modified, you must restart the Microsoft .NET instance (the underlying process). For the ASP.NET instance, you should run the IISRESET command.

The ActivityCollector.xml file is located in the `<i3_root>\products\dotnet\config directory`. Its settings affect all instances on the monitored server.

If you need to define instance-specific settings, for example for the Tracker threshold, place a copy of this file in the `<i3_root>\products\dotnet\config\instance name` directory and modify the settings as required.

The Aggregator and Tracker settings in this file then overrides the settings that are specified in the file that is located in the parent directory.

Example of the ActivityCollector.xml file

The following is an example for the ActivityCollector.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<activity-collector-config>
    <!-- psdn_dncol_act.exe configuration, configurable per server(default) & per instance -->
    <aggregator>
        <topnsql>5</topnsql><!-- Top N SQL statement to monitor -->
        <sla>5000</sla><!-- Red SLA breach value for top http invocations in msec. -->
        <nearsla>1000</nearsla><!-- Yellow SLA breach value for top http invocations in msec. -->
        <insane-rt>300</insane-rt><!-- Long running thread timeout in seconds -->
        <!-- The following two items are related to the Insight Smartlink feature and should not be altered by the user manually -->
        <bit-vector>false</bit-vector>
        <last-level-bit-vector>false</last-level-bit-vector>
    </aggregator>
    <!-- Tracker.dll settings, configurable per server (default) & per instance -->
    <tracker>
        <threshold>50</threshold><!-- Threshold in msec for filtering out events before forwarding it to the collector. -->
    </tracker>
</activity-collector-config>
```

About the ActivityCollector.xml file tags

The following table describes the important tags in the ActivityCollector.xml file. You may modify these tags only. It is not recommended to modify any other tags.

Table 13-2 Collector configuration file tags

Tag name	Description
activity-collector-config	The top-level XML tag.
Aggregator	The top-level definitions for the Collector agent's aggregator.
Topnsql	The top number of SQL statements to monitor.
Sla	The SLA (red) value (in milliseconds) for ASP.NET instance URLs.
nearsla	The `Near SLA` (yellow) value (in milliseconds) for ASP.NET instance URLs.
insane-rt	The timeout value for long running threads/URLs. A method or URL that is longer than this threshold is not collected.
tracker	Specific definitions for the tracker.
threshold	The threshold (in milliseconds) for filtering events before they are forwarded to the Collector agent.

Defining the DLLs to be monitored by using the Detection agent

A Microsoft .NET instance consists of the DLLs that make up your Microsoft .NET application. For Precise for Microsoft .NET to monitor a Microsoft .NET instance, you must first define the DLLs that you want to monitor.

You can use the Detection agent to detect all relevant DLLs that are currently running in the background. Using the Detection agent involves manually modifying the instrumentation configuration file.

To detect the DLLs currently running by using the Detection agent

1. Verify that your Microsoft .NET application is running.
2. Open the command prompt window.
3. Change directory to the `<i3_root>` directory.
4. Run the following command:
   - Microsoft .NET Framework version 1.1.
     `products\dotnet\install\psdn_detect_dlls.exe /im image name /f filter file name`
   - Microsoft .NET Framework version 2.0 and 3.0.
     `products\dotnet\install\FW2.0\psdn_detect_dlls.exe /im image name /f filter file name`

   where the parameter values should be set as follows:

   - **image name**

     If this is an ASP.NET instance , the name of this executable depends on the Internet Information Server (IIS) type.

     - IIS 6: `w3wp.exe`
     - IIS 7:

     If this is a regular .NET instance , the image name is the name of the .NET executable (without the path).

   - **filter file name**

     The path and name of the file that lists the modules to be filtered out, as follows:
     `products\dotnet\install\dlls_filter.xml`

   As alternative to this step, you can also use the Process Explorer. See http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx.
5. Save the XML output of the command to a temporary file.

To define the DLLs in the instrumentation file

1. Open the following file in an editor:
   `<i3_root>\products\dotnet\config\instrumentation.xml`
2. Within the XML output that you saved in step 5 above, locate the <module name> tag for a DLL file you want to instrument and copy the file name. For example, if you want to instrument the file petshop.web.dll, copy its name from the line `<module name="petshop.web.dll">`.
3. In the instrumentation file, paste the name of the DLL file into the entry for the instance that you want to monitor.
   For example:

```
<instances>
    <instance name="AspNetIIS6" >
        <dlls>
        <dll name="petshop.web.dll"/>
        ...
        </dlls>
    </instance>
</instances>
```

4. Repeat step 2 and step 3 for each DLL file that you want to instrument.
5. Restart your Microsoft .NET application for the changes to take effect.

Invoking the Instrumentation Driver utility

The instrumentation process is the execution of a Collector agent, which reads the code (the DLLs) of your application and stores an instrumented version in the `<i3_root>\products\dotnet\cache\instr` cache directory:

The contents of the cache can then be used for future invocations of your Microsoft .NET application, avoiding the time to re-instrument. The instrumentation process can last anywhere between a few minutes and an hour, depending on the size of the DLLs.

The instrumentation process consumes memory. The required memory for instrumenting a DLL is approximately 20 times the size of that DLL file. As a result, the startup of your Microsoft .NET application can be considerably slow.

The instrumentation is triggered when the Microsoft .NET application starts and one of the following criteria is applicable:.

- The .NET instance is installed for the first time.
- A change is made to the instrumentation.xml file (the instrumentation configuration file).
- A new code version is deployed on the monitored server.

To minimize the time it takes the Microsoft .NET application to start up, you can invoke a Precise for Microsoft .NET utility called Instrumentation Driver. The Instrumentation Driver populates the cache before it restarts the Microsoft .NET application.

If the instance uses DLLs from several directories or several single DLLs, invoke the Instrumentation Driver utility several times using the appropriate directory or DLL names.

It is recommended to stop your Microsoft .NET application before you invoke the Instrumentation Driver utility. If you invoke the utility while the application is running, the instrumentation results are placed in the cache and used only during the next startup of the Microsoft .NET application.

If your application changes, for example because a new version is deployed, or if you modify the instrumentation.xml file, you can activate the Instrumentation Driver utility to recalculate the cache contents.

The Instrumentation Driver utility may not calculate the cache contents for all DLLs that the application uses. These DLLs are only instrumented during the next startup of the Microsoft .NET application.

To invoke the Instrumentation Driver utility

1. After the Microsoft .NET instance is installed, configure the DLLs, classes, and methods to be instrumented in the instrumentation.xml file.
2. Open a command prompt window and run the following command:
   cd <*i3_root*>
   products\dotnet\bin\psdn_instr_validate.exe [-k *instance-name*]
   -f *directory-name* | -dll *name* [-gac]
   where:
   instance-name is the Microsoft .NET instance name, such as AspNetIIS5. If this parameter is omitted, the Instrumentation Driver works for all the Microsoft .NET instances that are installed on the monitored server.
   directory-name is the name of the DLL directory of the Microsoft .NET instance to be instrumented. This is the location where you deployed your Microsoft .NET code. For example, a DLL for an ASP.NET application may be stored in:
   `C:\Inetpub\wwwroot\MyWebApp\bin`
   name is the name of a single DLL (without the path). If this DLL is located in the GAC, add `-gac` when running the command.