Contract Evaluation Engine Extension Software Development Kit

Supports:

CIL/CEE Version 0.2.5.0

Dave Arnold

School of Computer Science

Carleton University

February 1st 2007

Abstract

The Contract Evaluation Engine Extension SDK provides a framework for creating user-defined static and dynamic checks. Such checks provide the basis for the construction of software contracts. Such contracts are used for software offshore outsourcing at a high-level, and for the implementation of the design-by-contract paradigm at a low-level. The SDK provides an API for performing static analysis against a compiled .NET assembly. The SDK also provides an API for analyzing and reporting on performance metrics gathered while executing the .NET assembly. The following document contains the complete API reference for both static and dynamic checks. Three tutorials for the creation of static and dynamic checks are provided.

Table of Contents

Changes

This section will briefly outline and track the changes made between this version of the Contract Evaluation Engine Extension Software Development Kit(SDK) and previous versions of the SDK. The changes listed include only major inclusions and modifications. Some changes will only be found in the document itself.

Old Version	Change Type	Change
0.2.0.0	Addition	The notion of contract extensions did not exist prior to version 0.2.5.0. As such all elements contained within this document are new.

Required Background

The Contract Evaluation Engine Extension SDK requires that the reader have a solid understanding of at least one .NET compatible language. The code examples presented in this document are written using Version 2.0 of the C# language or the C++/CLI language; however, any .NET compatible language can be used to write Contract Evaluation Engine Extensions.

In addition, knowledge of the Contract Intermediate Language (CIL) and the Contract Evaluation Engine (CEE) runtime is an asset, especially when developing performance extensions. For information on the CIL, see the CIL Specification document [1]. For information regarding the CEE runtime, see the Contract Editor Overview document [2].

The reader should also note that this SDK is for creating new static and performance extensions for use within the CEE. The Contract Language Extension SDK is used to create new high-level contract languages, which target the CIL. The Contract Language Extension SDK is not discussed in this document, but rather in a dedicated document [3].

System Requirements

In order to compile the examples shown in this document, the following tools will be required:

- Windows XP SP2, or Windows Vista
- Visual Studio 2005 .NET
  - The SDK cannot be used with prior versions of Visual Studio because the SDK libraries are compiled using features, which only exist in the 2005 version of the compilers. The entire visual studio package is not required for most of the examples; the free version of Visual C# 2005 Express Edition can also be used. It can be downloaded here:
  - http://msdn.microsoft.com/vstudio/express/visualcsharp/default.aspx
- .NET Framework 3.0 SDK (Not included Visual Studio)
  - The .NET Framework 3.0 SDK is not required for this version of the SDK; however, it will be required for Version 0.3.0.0, because it contains GUI features for Windows Vista. These features will be used to create the GUI in Version 0.3.0.0 of the Contract Editor, and the performance extensions will use them to write to the Contract Evaluation Report. The .NET Framework 3.0 SDK can be downloaded here:
  - http://msdn.microsoft.com/windowsvista/downloads/products/default.aspx
- The Contract Evaluation Engine Extension SDK Version 0.2.5.1
  - The SDK is included with this document.

Introduction

The Contract Evaluation Engine (CEE) supports the definition of user specified static and performance extensions. These extensions allow additional user-defined checks to be executed both during the static evaluation phase, and during analysis of metric information gathered by executing the Solution Under Test (SUT). To allow for maximum flexibility, there is no restriction on what an extension can examine or work with. For example, static extensions can actually modify the behavior of the SUT, so that a different performance extension can measure metrics, which could not be attained otherwise.

The Contract Evaluation Engine Extension SDK provides tools and examples to create your own extensions. All documentation regarding development of both static and performance extensions is contained within this document. The document includes details on the interfaces and code services available for extension developers. The document concludes with systematic examples of how to create both static and dynamic extensions.

The Contract Evaluation Engine Extension SDK is targeted towards static check and performance analysis developers, not the contract writer. The information contained within this document is technical in nature, and provides a method for independent software vendors (ISVs) to create their own set of specialized static checks and performance metric analysis tools.

All the examples contained within this document are shown in C#, with the exception of the second tutorial which is shown using C++/CLI. However, as all .NET languages are compatible, any language with supporting .NET bindings can be used to write static and performance extensions. The API illustrated in this document is based on the C# language, but can be imported into any .NET project, regardless of the implementation language.

Loading Extensions

All Contract Evaluation Engine Extensions are managed dlls, which are located in the "Extensions" sub-folder where the Contract Editor.exe is located. The Contract Editor will automatically examine the "Extensions" folder and all sub-folders contained within it for extensions. A dll file may contain any number of extensions. Non-extension dlls (support libraries, etc) can also be placed in the "Extensions" sub-folder, the Contract Editor will ignore them.

When the Contract Editor starts, it searches recursively through the "Extensions" sub-folder to for any managed dlls. Each managed dll is examined via reflection to determine of it contains one or more extensions. Once an extension is located it is initialized, and added to the Contract Editor's extension list. To determine if a given extension is discovered and initialized by the Contract Editor check the Log window. Figure 1, illustrates the Log window after loading three extensions: two static and one performance.

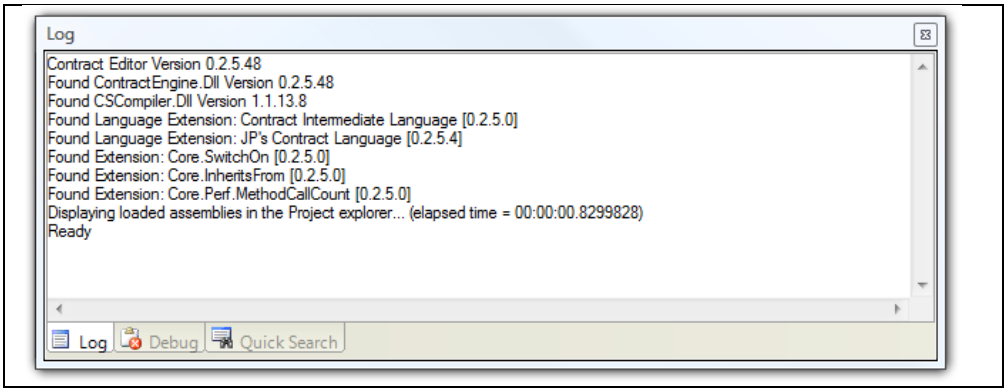


Figure 1 – Contract Editor Log Window

Contract Evaluation Extensions are shown by a "Found Extension:" prefix. The Log window displays the full extension name, including namespace, followed by the extension's version number. If an extension is not displayed in the Log window, then it has not been located by the Contract Editor and will not visible by the runtime.

Creating a Contract Evaluation Engine Extension

Extensions are created by creating a new class library that references the ExtensionAPI.Dll file. If you are creating a static extension, you may also want to reference the CSCompiler.Dll file as well. A user-defined class that inherits the IExtension interface defines an extension. Details of the IExtension interface will be presented in the following subsection.

IExtension

The IExtension interface is defined within the ExtensionAPI.Dll file, and is located in the "DaveArnold.Contracts.ExtensionAPI" namespace. Figure 2, illustrates the IExtension interface definition.

```
using System;

namespace DaveArnold.Contracts.ExtensionAPI
{
  public interface IExtension
  {
    Version Version { get; }
    string Name { get; }
    string Namespace { get; }
  }
}
```

Figure 2 – IExtension Interface Code Listing

The IExtension interface only defines three read-only properties which must be implemented in any user defined extension class. Each extension must be defined within its own class; however multiple extensions can be located within a single dll file. The following sections will examine each of the properties defined in the interface.

Version

The Version property is used to assign a version number to an extension. The version number is specified using the .NET Version class. The Version is specified by four integers: Major, Minor, Build, and Revision number. The value returned by the Version property has no meaning outside of the Contract Editor's Log window, but can be helpful with debugging. A common implementation of the Version property is to return the version number of the managed dll, which contains the extension. Figure 3 illustrates such an implementation. If you wish to specify a version number manually, the implementation shown in Figure 4 can be used.

```
public Version Version
{
        get
        {
                object[] attribs = System.Reflection.Assembly.GetAssembly(
                                this.GetType()).GetCustomAttributes(true);
                for (int i = 0; i < attribs.Length; i++)
                {
                        if (attribs[i] is AssemblyFileVersionAttribute)
                        {
                                AssemblyFileVersionAttribute av = attribs[i] as
                                        AssemblyFileVersionAttribute;
                                return new Version(av.Version);
                        }
                }
                return new Version(-1, -1, -1, -1);
        }
}
```

Figure 3 – Example Implementation of the Version Property (Dynamic)

```
public Version Version
{
        get
        {
                return new Version(0, 2, 5, 1);
        }
}
```

Figure 4 – Example Implementation of the Version Property (Literal)

Name

The extension name is used to identify the extension. Extension names are case sensitive. No two extensions defined within the same namespace may have the same name. If more than one extension is found to have the same name, the Contract Editor will only load the first extension located, and warnings will be displayed in the Log window as shown in Figure 5.

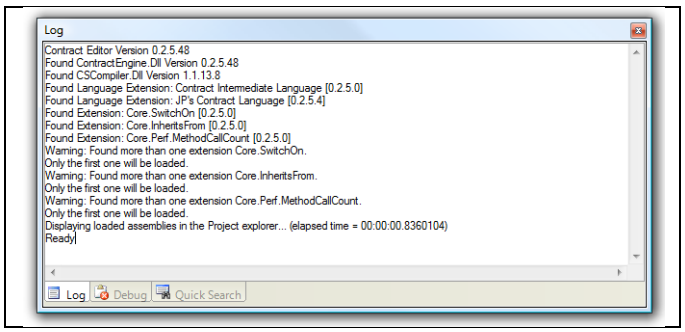


Figure 5 – Multiple Extension Warning

For static extensions, the extension name is used as the name used to call the extension. In the case of performance extensions, the extension name is only used when reporting the results on the Contract Evaluation Report. Usually, the implementation of the Name property consists of returning a string literal containing the name of the extension as shown in Figure 6.

```
public string Name
{
        get { return "InheritsFrom"; }
}
```

Figure 6 – Example Implementation of the Name Property

Namespace

The final property defined within the IExtension interface, is the Namespace property. The namespace property defines the namespace where the extension resides. Namespaces are separated by periods (.). The Namespace property must return the complete namespace where the extension is located. For example, if an extension were located in the "Perf" namespace which is a sub-namespace of "Core" then the Namespace property would return a string literal containing "Core.Perf". Like the Name property, the Namespace property is implemented by returning a string literal containing the namespace name, as shown in Figure 7.

```
public string Namespace
{
      get { return "Core"; }
}
```

Figure 7 – Example Implementation of the Namespace Property

These three properties make up the entirety of the IExtension interface. While these properties define location, name, and version of the extension, they do not define the entry point of the extension. That is, the interface does not define a method responsible for executing the extension. The reason for not having such a method defined in the IExtension interface is that static extensions and performance related extensions will require different parameters, as well; different static extensions will require a different number of parameters and parameter types. The following sections will examine how to create both static and performance extensions.

Static Extensions

Static extensions are called by code placed in the structure section of a contract. These extensions may take any number of parameters. To allow for such flexibility the class, which implements the IExtension interface, must define one or more static entry points. Static entry points are marked methods which are called by the Contract Evaluation Engine, with user supplied parameters. The static entry point is responsible for executing the extension's behavior and returning the result of that execution. The Contract Evaluation Engine Extension SDK provides an attribute class named StaticEntryPoint. The attribute can be applied to methods to indicate that the given method will be a static entry point. Figure 8, illustrates a signature for such an entry point.

```
[StaticEntryPoint]
public StaticEvaluationResult DoEvaluation(TypeContainer type, TypeContainer iface)
{
        // To Do: Put code to implement the static extension here
}
```

Figure 8 – Static Entry Point Signature

The entry point shown in Figure 8 begins with the application of the StaticEntryPoint attribute to the method. The attribute informs the Contract Evaluation Engine that this method can be used as an entry to the static extension defined by the Name property. The method shown has a return type of StaticEvaluationResult. All methods that are marked with the StaticEntryPoint attribute must have a return type of StaticEvaluationResult. If the method's return type is not the required type, an extension load error will be reported in the Contract Editor's Log window, and the entire extension will be unusable. Details regarding the StaticEvaluationResult type will follow. The name of the method, DoEvaluation in Figure 8, has no meaning and does not have to match the name of the extension. Finally, the method may take any number of parameters; however, the first parameter must be a TypeContainer. A TypeContainer is a class that represents a type defined within the SUT. Details of which will follow. The first parameter is required because all static extensions are called from within a contract. Each contract, by definition, is bound to a type defined within the SUT. It is this type, which is the value of the first TypeContainer parameter. In the case of Figure 8, the entry point also contains a second parameter of type TypeContainer. This would correspond to a static extension that would take a single parameter, which would be a type. Figure 9, shows a JPCL [5] code listing which calls a static extension named "InheritsFrom" which would have a static entry point like the one in Figure 8.

```
Using Core;
Contract Stack
{
        Belief StructureBelief("This belief applies to the entire structure");
        Structure
        {
                Belief Test("Generic Belief");
                Belief ~1("All stacks must inherit from IStack");
                {
                        InheritsFrom(IStack);
                }
        }
        Exports
        {
                Type IStack;
        }

}
```

Figure 9 – JPCL listing for calling a static extension named InheritsFrom

StaticEvaluationResult

As already discussed, all static extension entry point methods must return an instance of the StaticEvaluationResult class. As the name suggests, the class encapsulates the result of executing the

static extension. The StaticEvaluationResult class is defined within the ExtensionAPI.Dll file, and is located in the "DaveArnold.Contracts.ExtensionAPI" namespace. The class contains two constructors for creating a new StaticEvaluationResult Instance. The constructor definitions are shown in Figure 10.

```
public StaticEvaluationResult(StaticEvaluationResults result)
{
        // ...
}
 public StaticEvaluationResult(StaticEvaluationResults result, string text)
{
        // ...
}
```

Figure 10 – Constructor Definitions for the StaticEvaluationResult class

Both constructors take a value defined in the StaticEvaluationResults enumeration. The enumeration is also defined within the "DaveArnold.Contracts.ExtensionAPI" namespace and contains the values shown in Table 1. The second constructor also takes a string, which will be displayed on the Contract Evaluation Report only if the first parameter has a value of "Fail". That is, the string is used to specify why the "Fail" result was attained.

Value	Description
Pass	The static extension executed correctly and the requested check succeeded.
Fail	The static extension executed correctly and the requested check did not succeed.
Error	The static extension did not execute correctly (execution error).
Unknown	The result of executing the static extension cannot be determined. (This should never be used).

Table 1 – StaticEvaluationResults enumeration values

As shown in Table 1, the "Pass" and "Fail" values are used to indicate the result of calling the static extension. The "Error" value should only be used to indicate an error in executing the static extension (i.e. a programming error). The "Error" value should not be used to indicate that the static extension executed correctly, but the required check failed. Finally, the "Unknown" value is used to indicate that the StaticEvaluationResults object is in an inconsistent state, and should not be set from within a static extension.

Parameters

As already shown by Figure 8, the static entry point can contain any number of parameters. These parameters either map into literals specified in the contract languages or to types, methods, and field definitions, which map to a SUT counterpart. This section will examine each of the possible parameter types in detail. All static entry points must have at least one parameter. The first parameter for each static entry point must be of the TypeContainer type. It represents the type bound to the contract from which the static extension was called. Therefore, if a static entry point contains three parameters, it is representing a static extension that contains two parameter values. The first

parameter to the static entry point will be the containing type, while the second and third parameters will be the first and second parameters of the static extension call.

Boolean

The Boolean type represents either a true or a false value. To specify a static entry point that takes a parameter of the Boolean type, use the .NET System.Boolean structure. This structure is aliased by the *bool* keyword in most .NET languages. Figure 11, illustrates a static extension that takes one parameter of the Boolean type.

```
[StaticEntryPoint]
public StaticEvaluationResult DoEvaluation(TypeContainer type, bool bValue)
{
        // To Do: Put code to implement the static extension here
}
```

Figure 11 – Static Entry Point Signature with a Boolean Parameter

Class

A static extension may wish to operate and examine a type defined within the SUT. These types will be bound to an identifier contained within the contract language via the Contract Editor's Binding Tool [2]. These types are represented by the TypeContainer class [6]. The TypeContainer class is defined in the CSCompiler.dll file, which is included as part of the Contract Evaluation Engine Extension SDK. The TypeContainer class is located in the "DaveArnold.Contracts.CSCompiler" namespace.

The TypeContainer class is contained within the CSCompiler.dll file, rather than the ExtensionAPI.dll file, because the TypeContainer class represents a type defined within the SUT, and this type will be emitted by the C# compiler for profiling purposes. As the emission takes place following the execution of static checks, user defined static extensions may modify any aspect of the type. Such modification may include the addition of code to check pre- and post-conditions, adding specialized code to provide profiler instruction, and reporting tools for runtime analysis or debugging.

Note: Implementers of static extensions which modify the SUT should do so with caution. If a modification results in an error while recompiling the executable, the compile-time error will be presented to the user. Such an error will prevent the SUT from being profiled and the contract evaluation process will fail.

The following tables, list all of the relevant methods and properties defined within the TypeContainer type. The following tables intentionally omit a few methods defined in the TypeContainer type. The methods that are omitted should not be used by a static extension. These methods perform activities that should only be executed by the C# compiler once all of the static checks been completed. An example of such a method is the CloseType method, which will not allow any new elements to be added to the type. Some of the method, properties, and fields discussed below are based on the AST data types found in the Mono C# compiler [6].

AddClassOrStruct

<table>
<tr><td>Name:</td><td>AddClassOrStruct</td></tr>
<tr><td>Description:</td><td>Adds a new internal class, interface, or structure to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>TypeContainer</td><td>The class, interface, or structure to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>
The AddClassOrStruct method adds a new class, interface, or structure to the body of the current type. That is, the given class, interface, or structure is made an internal class, interface, or structure of the current type.

To create a new type container, you must create a class, interface or structure via the Class, Interface, or Struct types. To create a new class use the following constructor:

public Class(NamespaceEntry ns, TypeContainer parent,
MemberName name, int mod, Attributes attrs)

The first parameter specifies the namespace where the class is located. The second parameter specifies the enclosing type if any, null otherwise. The third parameter specifies the name of the class. The fourth parameter specifies the access modifiers for the class. Valid access modifiers are:

Modifiers.NEW
Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.PRIVATE
Modifiers.ABSTRACT
Modifiers.SEALED
Modifiers.UNSAFE

The final parameter specifies any attributes which are applied to the class.

To create a new interface use the following constructor:

public Interface(NamespaceEntry ns, TypeContainer parent,
MemberName name, int mod, Attributes attrs)

The first parameter specifies the namespace where the interface is located. The second parameter specifies the enclosing type if any, null otherwise. The third parameter specifies the name of the interface. The fourth parameter specifies the access modifiers for the interface. Valid access modifiers are:
</td></tr>
</table>

	Modifiers.NEW Modifiers.PUBLIC Modifiers.PROTECTED Modifiers.INTERNAL Modifiers.UNSAFE Modifiers.PRIVATE The final parameter specifies any attributes to be applied to the interface. To create a new structure use the following constructor: public Struct(NamespaceEntry ns, TypeContainer parent, MemberName name, int mod, Attributes attrs) The first parameter specifies the namespace where the structure is located. The second parameter specifies the enclosing type if any, null otherwise. The third parameter specifies the name of the structure. The fourth parameter specifies the access modifiers for the structure. Valid access modifiers are: Modifiers.NEW Modifiers.PUBLIC Modifiers.PROTECTED Modifiers.INTERNAL Modifiers.UNSAFE Modifiers.PRIVATE The final parameter specifies any attributes to be applied to the structure.

C# Example:	public static void AddClassToType(TypeContainer parent)
	{
	TypeContainer tc = new Class(parent.NamespaceEntry, parent,
	new MemberName("MyClass"),
	Modifiers.PUBLIC, null);
	parent.AddClassOrStruct(tc);
	}

Example Notes:	Creates a new class of the following form and adds it to the given type container.
	
	public class MyClass
	{
	}

Thread Safe:	No
C# Method Signature:	public bool AddClassOrStruct(TypeContainer c);
Version Information:	New in Version 0.2.5.1 of the SDK

AddConstant

<table>
<tr><td>Name:</td><td>AddConstant</td></tr>
<tr><td>Description:</td><td>Adds a new constant to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Const</td><td>The constant to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The AddConstant method adds a new constant to the current type.

Constants are represented by the Const class. The Const class contains a single constructor which will create a new constant. The constructor contains the following signature:

public Const(TypeContainer parent, Expression constant_type,
string name, Expression expr, int mod_flags,
Attributes attrs, Location loc)

The first parameter is the type where the constant will reside. The second parameter is an expression which defines the type of the constant. The third parameter is the name of the constant. The fourth parameter is a expression which defines the value of the constant. The fifth parameter specifies the access modifier flags for the constant. Valid access modifiers are:

Modifiers.NEW
Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.PRIVATE

The sixth parameter specifies any attributes which are to be applied to the constant. The final parameter specifies the source code location where the constant is located. When adding elements through a static extension, using the location of the parent element is recommended.</td></tr>
<tr><td>C# Example:</td><td>public static void AddIntConstantToType(TypeContainer parent)
{
Const c = new Const(parent,
new TypeExpression(typeof(int), parent.Location),
"NewConstant", new IntLiteral(10, parent.Location),
Modifiers.PUBLIC, null, parent.Location);
parent.AddConstant(c);
}</td></tr>
</table>

Example Notes:	Creates a new constant of the following form and adds it to the given type. public const int NewConstant = 10;
Thread Safe:	No
C# Method Signature:	public void AddConstant(Const constant);
Version Information:	New in Version 0.2.5.1 of the SDK

AddConstructor

<table>
<tr><td>Name:</td><td>AddConstructor</td></tr>
<tr><td>Description:</td><td>Adds a new constructor to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Constructor</td><td>The constructor to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>
The AddConstructor method adds a new constructor to the current type.

Constructors are represented by the Constructor class. The Constructor class contains a single constructor which will create a new constructor. The constructor contains the following signature:

public Constructor(TypeContainer ds, string name, int mod,
Parameters args, ConstructorInitializer init,
Location loc)

The first parameter is the type where the constructor will reside. The second parameter is the name of the constructor, which must match the container's name. The third parameter specifies the access modifier flags for the constructor. Valid access modifiers are:

Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.STATIC
Modifiers.UNSAFE
Modifiers.EXTERN
Modifiers.PRIVATE

The fourth parameter specifies any parameters that the constructor takes. The fifth parameter specifies any initialization the constructor performs. The final parameter specifies the source code location where the constructor is located. When adding elements using a static extension, using the location of the parent element is recommended.
</td></tr>
</table>

C# Example:	public static void AddConstructorToType(TypeContainer tc)
	{
	Parameters args = new Parameters(
	new Parameter[] {
	new Parameter(typeof(int), "i",
	Parameter.Modifier.NONE, null, tc.Location)
	});
	ArrayList inits = new ArrayList();
	ConstructorInitializer ci = new ConstructorInitializer(inits,
	tc.Location);
	Constructor c = new Constructor(tc, tc.MemberName.Name,
	Modifiers.PUBLIC, args, ci, tc.Location);
	tc.AddConstructor(c);
	}

Example Notes:	Creates a new constructor of the following form and adds it to the given type.
	
	public MyType(int i)
	{
	}

Thread Safe:	No
C# Method Signature:	public void AddConstructor(Constructor constructor);
Version Information:	New in Version 0.2.5.1 of the SDK

AddDelegate

<table>
<tr><td>Name:</td><td>AddDelegate</td></tr>
<tr><td>Description:</td><td>Adds a new delegate to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Delegate</td><td>The delegate to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>
The AddDelegate method adds a new delegate to the current type.

Delegates are represented by the Delegate class. The Delegate class contains a single constructor which will create a new delegate. The constructor contains the following signature:

public Delegate(NamespaceEntry ns, TypeContainer parent,
Expression type, int mod_flags, MemberName name,
Parameters param_list, Attributes attrs)

The first parameter is the namespace where the delegate will reside. The second parameter specifies the parent of the delegate. The parameter value should be the same as the type container, to which the delegate is being added. The third parameter specifies the return type of the delegate. The fifth parameter specifies the access modifier flags for the delegate. Valid access modifiers are:

Modifiers.NEW
Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.UNSAFE
Modifiers.PRIVATE

The sixth parameter specifies the delegate's name. The seventh parameter specifies any parameters which the delegate requires. Finally, the seventh parameter specifies any attributes which are to be applied to the delegate.
</td></tr>
</table>

C# Example:	public static void AddDelegateToType(TypeContainer parent)
	{
	Parameters args = new Parameters(
	new Parameter[] {
	new Parameter(typeof(int), "i",
	Parameter.Modifier.NONE, null,
	parent.Location),
	new Parameter(typeof(char), "c",
	Parameter.Modifier.REF, null,
	parent.Location)
	}
	);
	Delegate d = new Delegate(parent.NamespaceEntry, parent,
	new TypeExpression(typeof(bool)), Modifiers.PUBLIC,
	new MemberName("MyDelegate"), args, null);
	parent.AddDelegate(d);
	}

Example Notes:	Creates a new delegate of the following form and adds it to the given type. public delegate bool MyDelegate(int i, ref char c);
Thread Safe:	No
C# Method Signature:	public void AddDelegate(Delegate delegate);
Version Information:	New in Version 0.2.5.1 of the SDK

AddEnum

Name:	**AddEnum**
Description:	**Adds a new enumeration to this type**
Parameters:	1
Parameter Values:	Ordinal: 1 — Type: Enum — Description: The enumeration to add

Remarks:	The *AddEnum* method adds a new enumeration to the current type.
	
	Enumerations are represented by the Enum class. The Enum class contains a single constructor which will create a new enumeration. The constructor contains the following signature:
	
	`public Enum(NamespaceEntry ns, TypeContainer parent, Expression type, int mod_flags, MemberName name, Attributes attrs)`
	
	The first parameter specifies the namespace where the enumeration will reside. The second parameter specifies the parent of the enumeration. The paremeter value should be the same as the type container, to which the enumeration is being added. The third parameter is the enumeration's type. The fourth parameter specifies the access modifier flags for the enumeration. Valid access modifiers are:
	
	`Modifiers.NEW`
	`Modifiers.PUBLIC`
	`Modifiers.PROTECTED`
	`Modifiers.INTERNAL`
	`Modifiers.PRIVATE`
	
	The fifth parameter specifies the name of the eumeration. The final parameter specifies any attributes which are to be applied to the enumeration.

C# Example:	`public static void AddEnumToType(TypeContainer parent)`
	`{`
	`  Enum e = new Enum(parent.NamespaceEntry, parent,`
	`  new TypeExpression(typeof(int), parent.Location),`
	`  Modifiers.PUBLIC, new MemberName("MyEnum"), null);`
	
	`  EnumMember first = new EnumMember(e, null, null,`
	`  new MemberName("First"), null);`
	`  EnumMember second = new EnumMember(e, first, null,`
	`  new MemberName("Second"), null));`
	
	`  e.AddEnumMember(first);`
	`  e.AddEnumMember(second);`
	`  parent.AddEnum(e);`
	`}`

Example Notes:	Creates a new enumeration of the following form and adds it to the given type.
	
	`public enum MyEnum`
	`{`
	`First,`
	`Second`
	`}`

Thread Safe:	No
C# Method Signature:	`public void AddEnum(Enum enum);`
Version Information:	New in Version 0.2.5.1 of the SDK

AddEvent

<table>
<tr><td>Name:</td><td>AddEvent</td></tr>
<tr><td>Description:</td><td>Adds a new event to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Event</td><td>The event to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The AddEvent method adds a new event to the current type.

Events are represented by the Event class. The Event class is abstract. For the purposes of adding an event to a type, the EventField class can be used. The EventField class contains a single constructor, which will create a new event. The constructor contains the following signature:

public EventField(TypeContainer parent, Expression type,
int mod_flags, bool is_iface, MemberName name,
Attributes attrs)

The first parameter specifies the parent where the event will reside. The second parameter specifies the event's type (a delegate). The third parameter specifies the access modifier flags for the event. Valid access modifiers are:

Modifiers.NEW
Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.PRIVATE
Modifiers.STATIC
Modifiers.VIRTUAL
Modifiers.SEALED
Modifiers.OVERRIDE
Modifiers.UNSAFE
Modifiers.ABSTRACT

The fourth parameter indicates if the event is being defined as an interface (i.e. within an interface). The fifth parameter specifies the name of the event. The final parameter specifies any attributes which are to be applied to the event.</td></tr>
<tr><td>C# Example:</td><td>public static void AddEventToType(TypeContainer parent)
{
Event e = new Event(parent, typeof(EventHandler),
Modifiers.PUBLIC, false, new MemberName("MyEvent"), null);

parent.AddEvent(e);
}</td></tr>
</table>

Example Notes:	Creates a new event of the following form and adds it to the given type. public event EventHandler MyEvent;
Thread Safe:	No
C# Method Signature:	public void AddEvent(Event event);
Version Information:	New in Version 0.2.5.1 of the SDK

AddField

Name:	**AddField**
Description:	**Adds a new field to this type**
Parameters:	1
Parameter Values:	Ordinal: 1 — Type: FieldMember — Description: The field to add
Remarks:	The *AddField* method adds a new field to the current type. Fields are represented by the Field class, which implements the abstract FieldMember class. The Field class contains a single constructor, which will create a new field. The constructor contains the following signature: `public Field(TypeContainer parent, Expression type, int mod, string name, Attributes attrs, Location loc)` The first parameter specfies the parent where the field will reside. The second parameter specifies the field's type. The third parameter specifies the access modifier flags for the field. Valid access modifiers are: Modifiers.NEW Modifiers.PUBLIC Modifiers.PROTECTED Modifiers.INTERNAL Modifiers.PRIVATE Modifiers.STATIC Modifiers.VOLATILE Modifiers.UNSAFE Modifiers.READONLY The fourth parameter specifies the name of the field. The fifth parameter specifies any attributes which are to be applied to the field. The final parameter specifies the source code location where the field is located. When adding elements programically, using the location of the parent element is recommended.

C# Example:	`public static void AddFieldToType(TypeContainer parent)`
	`{`
	`  Field f = new Field(parent,`
	`  new TypeExpression(typeof(string), parent.Location),`
	`  Modifiers.PRIVATE, "MyField", null, parent.Location);`
	`  parent.AddField(f);`
	`}`

Example Notes:	Creates a new field of the following form and adds it to the given type. `private string MyField;`
Thread Safe:	No
C# Method Signature:	`public void AddField(FieldMember field);`
Version Information:	New in Version 0.2.5.1 of the SDK

AddIndexer

Name:	**AddIndexer**
Description:	**Adds a new indexer to this type**
Parameters:	1
Parameter Values:	Ordinal: 1 — Type: Indexer — Description: The indexer to add

Remarks:	The *AddIndexer* method adds a new indexer to the current type.
	
	Indexers are represented by the Indexer class. The Indexer class contains a single constructor, which will create a new indexer. The constructor contains the following signature:
	
	`public Indexer(TypeContainer ds, Expression type, MemberName name, int mod, bool is_iface, Parameters parameters, Attributes attrs, Accessor get_block, Accessor set_block)`
	
	The first parameter specifies the parent where the indexer will reside. The second parameter specifies the indexer's return type. The third parameter denotes the indexer's name. The fourth parameter specifies the access modifier flags for the indexer. Valid access modifiers are:
	
	`Modifiers.NEW`
	`Modifiers.PUBLIC`
	`Modifiers.PROTECTED`
	`Modifiers.INTERNAL`
	`Modifiers.PRIVATE`
	`Modifiers.VIRTUAL`
	`Modifiers.SEALED`
	`Modifiers.OVERRIDE`
	`Modifiers.UNSAFE`
	`Modifiers.EXTERN`
	`Modifiers.ABSTRACT`
	
	The fifth parameter indicates if the indexer is being defined within an interface or not. The sixth parameter contains a list of any of the indexer's parameters. The seventh parameter specifies any attributes which are to be applied to the indexer. The final two parameters specifiy the get and set accessor blocks contained within the indexer, respectively. Accessors may be created by creating instances of the Accessor class. The Accessor class contains one constructor with the following signature:
	
	`public Accessor(ToplevelBlock b, int mod, Attributes attrs, Location loc)`

	The first parameter specifies the top instruction block, which will make up the body of the accessor. For more information on method bodies, please see the discussion on the Method parameter type later in this document. The second parameter specifies the access modifier flags for the indexer. Valid access modifiers are: Modifiers.PUBLIC Modifiers.PROTECTED Modifiers.INTERNAL Modifiers.PRIVATE The third parameter specifies any attributes which are to be applied to the accessor. The final parameter specifies the source code location where the field is located. When adding elements in a static extension, using the location of the parent element is recommended.

C# Example:	public static void AddIndexerToType(TypeContainer parent)
	{
	Location loc = parent.Location;
	Parameter p = new Parameter(
	new TypeExpression(typeof(int), loc), "idx",
	Parameter.Modifier.NONE, null, loc);
	Parameters args = new Parameters(new Parameter[] { p });
	ToplevelBlock block = new ToplevelBlock(args, loc);
	ArrayList callArgs = new ArrayList();
	callArgs.Add(new Argument(
	new ParameterReference(p, block, 0, loc),
	Argument.AType.Expression));
	block.AddStatement(new Return(new ElementAccess(new
	FieldExpr(parent.FindBaseMemberWithSameName(
	"theArray", true) as FieldInfo, loc), callArgs), loc));
	Accessor get = new Accessor(block, Modifiers.PUBLIC,
	null, parent.Location);
	Indexer indexer = new Indexer(parent,
	new TypeExpression(typeof(string), loc),
	new MemberName("this"), Modifiers.PUBLIC,
	false, args, null, get, null);
	parent.AddIndexer(indexer);
	}

Example Notes:	Creates a new indexer with a get accessor of the following form and adds it to the given type. The example assumes that the containing type defines a field called "theArray", which supports the "[]" operator. public string this[int idx] { get { return theArray[idx]; } }
Thread Safe:	No
C# Method Signature:	public void AddIndexer(Indexer indexer);
Version Information:	New in Version 0.2.5.1 of the SDK

AddInterface

Name:	**AddInterface**
Description:	**Adds a new interface to this type**
Parameters:	1
Parameter Values:	Ordinal: 1 — Type: TypeContainer — Description: The interface to add

Remarks:	The *AddInterface* method adds a new interface to the current type.
	
	Interfaces are represented by the Interface class, which implements the abstract TypeContainer class. The Interface class contains a single constructor which will create a new interface. The constructor contains the following signature:
	
	`public Interface(NamespaceEntry ns, TypeContainer parent, MemberName name, int mod, Attributes attrs)`
	
	The first parameter specifies the namespace where the interface will reside. The second parameter is the containing type which will hold the interface. The third parameter is the interface's name. The fourth parameter specifies the access modifier flags for the interface. Valid access modifiers are:
	
	`Modifiers.NEW`
	`Modifiers.PUBLIC`
	`Modifiers.PROTECTED`
	`Modifiers.INTERNAL`
	`Modifiers.UNSAFE`
	`Modifiers.PRIVATE`
	
	The final parameter specifies any attributes which are to be applied to the interface.

C# Example:	`public static void AddInterfaceToType(TypeContainer parent)`
	`{`
	`  Interface iface = new Interface(parent.NamespaceEntry,`
	`  parent, new MemberName("MyInterface"),`
	`  Modifiers.PUBLIC, null);`
	`  parent.AddInterface(iface);`
	`}`

Example Notes:	Creates a new interface of the following form and adds it to the given type. As the Interface class inherits from the TypeContainer class, any of the operations discussed in this section can be used to add members to the interface. `public interface MyInterface { }`
Thread Safe:	No
C# Method Signature:	`public bool AddInterface(Interface interface);`
Version Information:	New in Version 0.2.5.1 of the SDK

AddIterator

<table>
<tr><td>Name:</td><td>AddIterator</td></tr>
<tr><td>Description:</td><td>Adds a new iterator to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Iterator</td><td>The iterator to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The AddIterator method adds a new iterator to the current type.

Iterators are represented by the Iterator class. Iterators are a new feature in C# 2.0. An iterator is a method, get accessor, or operator that enables you to support “foreach” iteration, without having to implement the entire “IEnumerable” interface. Instead just the iterator is provided, the iterator traverses the data structures in your class. The compiler will automatically generate “Current”, “MoveNext”, and “Dispose” methods. The Iterator class contains a single constructor, which will create a new iterator. The constructor contains the following signature:

public Iterator(IMethodData m_container,
TypeContainer container, int modifiers)

The first parameter specifies the method body for the iterator. The second parameter is the containing type which will hold the iterator. The fourth parameter specifies the access modifier flags for the iterator. Valid access modifiers are:

Modifiers.NEW
Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.PRIVATE
Modifiers.ABSTRACT
Modifiers.SEALED
Modifiers.UNSAFE</td></tr>
<tr><td>C# Example:</td><td>public static void AddIteratorToType(TypeContainer parent,
IMethodData method) {
Iterator iter = new Iterator(method, parent, Modifiers.PUBLIC);
iter.DefineIterator();
parent.AddIterator(iter); }</td></tr>
<tr><td>Example Notes:</td><td>Creates a new iterator using the given method body, calls “DefineIterator” to create the compiler generated iterator methods, and finaly adds the iterator to the parent type.

For information on creating and working with method bodies, please see the Method parameter type section later in this document.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public void AddIterator(Iterator iterator);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

AddMethod

<table>
<tr><td>Name:</td><td>AddMethod</td></tr>
<tr><td>Description:</td><td>Adds a new method to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Method</td><td>The method to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>
The AddMethod method adds a new method to the current type.

Methods are represented by the Method class. The Method class contains a single constructor, which will create a new method. The constructor contains the following signature:

public Method(TypeContainer ds, Expression return_type,
int mod, bool is_iface, MemberName name,
Parameters parameters, Attributes attrs)

The first parameter specifies the type that will contain the method. The second parameter is an expression which will denote the method's return type. If the return type is void, a "null" value can be used. The third parameter specifies the access modifier flags for the method. Valid access modifiers are:

Modifiers.NEW
Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.PRIVATE
Modifiers.STATIC
Modifiers.VIRTUAL
Modifiers.SEALED
Modifiers.OVERRIDE
Modifiers.ABSTRACT
Modifiers.UNSAFE
Modifiers.METHOD_YIELDS
Modifiers.EXTERN

The fourth parameter indicates if the method is being defined within an interface. The fifth parameter specifies the name of the method. The sixth parameter denotes any parameters the method takes. The final parameter specifies any attributes which are to be applied to the method.
</td></tr>
</table>

C# Example:	public static void AddMethodToType(TypeContainer parent)
	{
	Parameters param = new Parameters(new Parameter[] {
	new Parameter(new TypeExpression(typeof(int),
	parent.Location), "i", Parameter.Modifier.NONE,
	null, parent.Location)
	});
	Method m = new Method(parent,
	new TypeExpression(typeof(int), parent.Location),
	Modifiers.PUBLIC, false,
	new MemberName("MyMethod"), param, null);
	ToplevelBlock block = new ToplevelBlock(param,
	parent.Location);
	block.AddStatement(new Return(new
	ParameterReference(param.GetParameterByName("i"),
	block, 0, parent.Location), parent.Location));
	m.Block = block;
	
	parent.AddMethod(m);
	}

Example Notes:	Creates a new method of the following form and adds it to the given type. For more information on method bodies, please see the Method parameter type section later in this document.
	
	public int MyMethod(int i)
	{
	return i;
	}

Thread Safe:	No
C# Method Signature:	public void AddMethod(Method method);
Version Information:	New in Version 0.2.5.1 of the SDK

AddOperator

<table>
<tr><td>Name:</td><td>AddOperator</td></tr>
<tr><td>Description:</td><td>Adds a new operator to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Operator</td><td>The operator to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>
The AddOperator method adds a new operator to the current type.

Operators are represented by the Operator class. The Operator class contains a single constructor, which will create a new operator. The constructor contains the following signature:

public Operator(TypeContainer parent, OpType type,
Expression ret_type, int mod_flags,
Parameters parameters, ToplevelBlock block,
Attributes attrs, Location loc)

The first parameter specifies the type that will contain the operator. The second parameter indicates the type of operator being specified. The third parameter is an expression which will denote the operator's return type. The fourth parameter specifies the access modifier flags for the operator. Valid access modifiers are:

Modifiers.PUBLIC
Modifiers.UNSAFE
Modifiers.EXTERN
Modifiers.STATIC

The fifth parameter denotes any parameters the operator takes. The sixth parameter specifies the top level method block that defines the body of the operator. The seventh parameter specifies any attributes which are to be applied to the operator. The final parameter specifies the source code location where the operator is located. When adding elements in a static extension, using the location of the parent element is recommended.
</td></tr>
</table>

C# Example:	public static void AddOperatorToType(TypeContainer parent)
	{
	Location loc = parent.Location;
	Parameter a = new Parameter(
	new TypeExpression(typeof(int), loc),
	"a", Parameter.Modifier.NONE, null, loc);
	Parameter b = new Parameter(
	new TypeExpression(typeof(int), loc),
	"b", Parameter.Modifier.NONE, null, loc);
	Parameters param = new Parameters(new Parameter[] {a, b });
	
	ToplevelBlock block = new ToplevelBlock(param, loc);
	block.AddStatement(new Return(
	new Binary(Binary.Operator.Addition,
	new ParameterReference(a, block, 0, loc),
	new ParameterReference(b, block, 0, loc)), loc));
	
	Operator op = new Operator(parent, OpType.Addition,
	new TypeExpression(typeof(int), loc),
	Modifiers.PUBLIC | Modifiers.STATIC,
	param, block, null, loc);
	
	parent.AddOperator(op);
	}

Example Notes:	Creates a new operator of the following form and adds it to the given type. For more information on method bodies, please see the Method parameter type section later in this document.
	
	public static int operator +(int a, int b)
	{
	return a + b;
	}

Thread Safe:	No
C# Method Signature:	public void AddOperator(Operator operator);
Version Information:	New in Version 0.2.5.1 of the SDK

AddPart

<table>
<tr><td>Name:</td><td>AddPart</td></tr>
<tr><td>Description:</td><td>Adds a new class part to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>ClassPart</td><td>The class part to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>
The AddPart method adds a new class part to the current type. Class parts allow for the addition of additional partial classes to an existing partial class.

Class parts are represented by the ClassPart class. The ClassPart class contains a single constructor, which will create a new class part. The constructor contains the following signature:

public ClassPart(NamespaceEntry ns, PartialContainer pc,
TypeContainer parent, int mod, Attributes attrs,
Kind kind)

The first parameter specifies the namespace where the partial type will reside. The second parameter specifies the partial container to which this class part is being added to. The third parameter specifies the type that will contain the partial type. The fourth parameter specifies the access modifier flags for the class part. Valid access modifiers are:

Modifiers.NEW
Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.PRIVATE
Modifiers.ABSTRACT
Modifiers.SEALED
Modifiers.UNSAFE

The fifth parameter specifies any attributes which are to be applied to the class part. The final parameter specifies the type of type container being added. Valid types are:

Kind.Root
Kind.Struct
Kind.Class
Kind.Interface

A second method to create a new class part is via the use of the static “CreatePart” method defined in the PartialContainer class. The “CreatePart” method has the following signature:
</td></tr>
</table>

	public static ClassPart CreatePart(NamespaceEntry ns, TypeContainer parent, MemberName name, int mod, Attributes attrs, Kind kind, Location loc) The first parameter specifies the namespace where the partial type will reside. The second parameter specifies the parent where the partial type will be created. The third parameter specifies the name of the partial type. The fourth parameter specifies the access modifier flags for the class part. Valid access modifiers are: Modifiers.NEW Modifiers.PUBLIC Modifiers.PROTECTED Modifiers.INTERNAL Modifiers.PRIVATE Modifiers.ABSTRACT Modifiers.SEALED Modifiers.UNSAFE The fifth parameter specifies any attributes which are to be applied to the operator. The sixth parameter specifies the type of type container being added. Valid types are: Kind.Root Kind.Struct Kind.Class Kind.Interface The final parameter specifies the source code location where the class part is located. When adding elements in a static extension, using the location of the parent element is recommended. Depending on the programming language used to implement the SUT, the notion of partial classes may not be supported. In this case, there will obviously not be any partial class definitions found within the SUT. However, because the SUT's executable code is translated into C# it would be possible for a static extension to add a partial class to the SUT if desired.

C# Example:	public static void AddPartToType(TypeContainer parent)
	{
	ClassPart classPart =
	PartialContainer.CreatePart(parent.NamespaceEntry, parent,
	new MemberName("MyPartialClass"),
	Modifiers.PUBLIC, null, Kind.Class, parent.Location);
	
	ClassPart classPart2 =
	PartialContainer.CreatePart(parent.NamespaceEntry, parent,
	new MemberName("MyPartialClass"),
	Modifiers.PUBLIC, null, Kind.Class, parent.Location);
	
	parent.AddPart(classPart);
	parent.AddPart(classPart2);
	}

Example Notes:	Creates two new class parts, each of the following form, and adds them to the given type. Each class part is subclassed from the TypeContainer class, so all of the operations listed in this section can be used to add elements to the partial classes.
	
	public partial class MyPartialClass
	{
	}

Thread Safe:	No
C# Method Signature:	public void AddPart(ClassPart part);
Version Information:	New in Version 0.2.5.1 of the SDK

AddProperty

Name:	**AddProperty**
Description:	**Adds a new property to this type**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>Property</td><td>The property to add</td></tr></table>
Remarks:	The *AddProperty* method adds a new property to the current type. Properties are represented by the Property class. The Property class contains a single constructor, which will create a new property. The constructor contains the following signature: `public Property(TypeContainer ds, Expression type, int mod, bool is_iface, MemberName name, Attributes attrs, Accessor get_block, Accessor set_block)` The first parameter specifies the type to which the property will be added. The second parameter denotes the return type for the property. The third parameter specifies the access modifier flags for the property. Valid access modifiers are: Modifiers.NEW Modifiers.PUBLIC Modifiers.PROTECTED Modifiers.INTERNAL Modifiers.PRIVATE Modifiers.STATIC Modifiers.SEALED Modifiers.OVERRIDE Modifiers.ABSTRACT Modifiers.UNSAFE Modifiers.EXTERN Modifiers.METHOD_YIELDS Modifiers.VIRTUAL The fourth parameter indicates if the property declaration is an interface or not. The fifth parameter specifies the property's name. The sixth parameter specifies any attributes which are to be applied to the property. Depending on the programming language used to implement the SUT, the notion of properties may not be supported. In this case, there will obviously not be any property definitions found within the SUT. However, because the SUT's executable code is translated into C# it would be possible for a static extension to add a property to the SUT if desired.

C# Example:	public static void AddPropertyToType(TypeContainer parent)
	{
	Location loc = parent.Location;
	ToplevelBlock block = new ToplevelBlock(loc);
	block.AddStatement(new Return(
	new BoolLiteral(true, loc), loc));
	Accessor get = new Accessor(block, Modifiers.PUBLIC, null, loc);
	Property prop = new Property(parent,
	new TypeExpression(typeof(bool), loc),
	Modifiers.PUBLIC, false,
	new MemberName("MyProperty"), null, get, null);
	
	parent.AddProperty(prop);
	}

Example Notes:	Creates a new property with a get accessor of the following form, and adds them to the given type. public bool MyProperty { get { return true; } }
Thread Safe:	No
C# Method Signature:	public void AddProperty(Property property);
Version Information:	New in Version 0.2.5.1 of the SDK

AddType

<table>
<tr><td>Name:</td><td>AddType</td></tr>
<tr><td>Description:</td><td>Adds a new type to this type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>TypeContainer</td><td>The type to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>
The AddType method adds a new type to the body of the current type. That is, the given type is made an internal class, interface, or structure of the current type.

To create a new type container, you must create a class, interface or structure via the Class, Interface, or Struct types. To create a new class use the following constructor:

public Class(NamespaceEntry ns, TypeContainer parent,
MemberName name, int mod, Attributes attrs)

The first parameter specifies the namespace where the class is located. The second parameter specifies the enclosing type if any, null otherwise. The third parameter specifies the name of the class. The fourth parameter specifies the access modifiers for the class. Valid access modifiers are:

Modifiers.NEW
Modifiers.PUBLIC
Modifiers.PROTECTED
Modifiers.INTERNAL
Modifiers.PRIVATE
Modifiers.ABSTRACT
Modifiers.SEALED
Modifiers.UNSAFE

The final parameter specifies any attributes which are applied to the class.

To create a new interface use the following constructor:

public Interface(NamespaceEntry ns, TypeContainer parent,
MemberName name, int mod, Attributes attrs)

The first parameter specifies the namespace where the interface is located. The second parameter specifies the enclosing type if any, null otherwise. The third parameter specifies the name of the interface. The fourth parameter specifies the access modifiers for the interface. Valid access modifiers are:
</td></tr>
</table>

	Modifiers.NEW Modifiers.PUBLIC Modifiers.PROTECTED Modifiers.INTERNAL Modifiers.UNSAFE Modifiers.PRIVATE The final parameter specifies any attributes to be applied to the interface. To create a new structure use the following constructor: public Struct(NamespaceEntry ns, TypeContainer parent, MemberName name, int mod, Attributes attrs) The first parameter specifies the namespace where the structure is located. The second parameter specifies the enclosing type if any, null otherwise. The third parameter specifies the name of the structure. The fourth parameter specifies the access modifiers for the structure. Valid access modifiers are: Modifiers.NEW Modifiers.PUBLIC Modifiers.PROTECTED Modifiers.INTERNAL Modifiers.UNSAFE Modifiers.PRIVATE The final parameter specifies any attributes to be applied to the structure.

C# Example:	public static void AddTypeToType(TypeContainer parent)
	{
	TypeContainer tc = new Class(parent.NamespaceEntry, parent,
	new MemberName("MyClass"),
	Modifiers.PUBLIC, null);
	parent.AddType(tc);
	}

Example Notes:	Creates a new class of the following form and adds it to the given type container.
	
	public class MyClass
	{
	}

Thread Safe:	Yes
C# Method Signature:	public void AddType(TypeContainer c);
Version Information:	New in Version 0.2.5.1 of the SDK

AppendMethod

<table>
<tr><td>Name:</td><td>AppendMethod</td></tr>
<tr><td>Description:</td><td>Adds a new method to this type after the type has been closed</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Method</td><td>The method to add</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The AppendMethod method adds a new method to the current type after the type has been closed.

This method should not be called by user code. Use the previously discussed AddMethod method, to add a new method to a type. The AppendMethod method is used by iterators to add the compiler generated methods to a type.</td></tr>
<tr><td>C# Example:</td><td>N/A</td></tr>
<tr><td>Example Notes:</td><td>N/A</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public void AppendMethod(Method method);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

AsAccessible

<table>
<tr><td>Name:</td><td>AsAccessible</td></tr>
<tr><td>Description:</td><td>Determines if the given type is as accessible as the member defined by the given accessibility flags</td></tr>
<tr><td>Parameters:</td><td>2</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Type</td><td>The type to check against</td></tr>
<tr><td>2</td><td>Integer</td><td>The member's flags to check against the type</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The AsAccessible method is used to answer the question: "is the given type as accessible as a member?" The member's accessibility is defined by the second parameter.

AsAccessible is implemented by testing that every place where the member is accessible, that the given type is also accessible. That is, given that the member is denoted by M, and the type is denoted by P, AsAccessible returns true if the following expression holds:
~ (M -> P) == 0 <-> ~ (~M | P) == 0</td></tr>
<tr><td>C# Example:</td><td>public static bool IsAsAccessible(TypeContainer parent, Type type)
{
return parent.AsAccessible(type, parent.ModFlags);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method returns true if the given type is as accessible as the type container. The type container's accessibility is specified by the accessibility flags, which can be accessed via the ModFlags property.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public bool AsAccessible(Type p, int flags)</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

CheckAccessLevel

Name:	**CheckAccessLevel**
Description:	**Ensures that the access level of the given type has valid accessibility modifiers**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>Type</td><td>The type to check accessibility for</td></tr></table>
Remarks:	The *CheckAccessLevel* method is used to ensure that valid access modifiers have been placed on the given type. The *CheckAccessLevel* method should be called when new access modifiers are being applied to an existing type, or a new type is being added to the SUT. If valid access modifiers are specified on the given type, *CheckAccessLevel* will return true. A value of false will be returned otherwise. Valid access modifiers consist of any combination of access modifiers which are valid according to the C# specification. For example, a private static method is an invalid access modifier combination.

C# Example:	`public static bool CheckAccessLevelForType(TypeContainer parent,`
	`    Type t)`
	`{`
	`    return parent.CheckAccessLevel(t);`
	`}`

Example Notes:	The example method returns true if the given type contains valid access modifier flags. If the access modifier flags are invalid, then a value of false is returned.
Thread Safe:	No
C# Method Signature:	`public bool CheckAccessLevel(Type check_type)`
Version Information:	New in Version 0.2.5.1 of the SDK

CheckObsoleteness

Name:	**CheckObsoleteness**
Description:	**Checks to see if the containing type is marked as obsolete**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>Location</td><td>The source code location where the check is based</td></tr></table>
Remarks:	The *CheckObsoleteness* method is used to check if the containing type container is marked as obsolete. Marking a type as obsolete consists of using the "Obsolete" attribute before the class definition. The *CheckObsoleteness* method will also examine base types. That is, if the containing type container is not marked as obsolete, but one of its base types is marked as obsolete, the method will report an obsolete warning message. The *CheckObsoleteness* method uses the C# compiler error reporting mechanism to indicate if a type is marked as obsolete or not. The example, will present a method of creating a Boolean wrapper method. Obsolete program elements will issue a compile time warning when they are referenced in code. The idea is to mark a given program element as obsolete to discourage its usage and to indicate to the user that such element should no longer be used.

C# Example:	`public static bool IsObsolete(TypeContainer parent)`
	`{`
	`  bool result = false;`
	`  int warnCount = Report.Warnings;`
	`  System.IO.TextWriter writer = Report.Stderr;`
	`  Report.Stderr = null;`
	`  parent.CheckObsoleteness(parent.Location);`
	`  if (Report.Warnings != warnCount)`
	`  {`
	`  result = true;`
	`  Report.Warnings = warnCount;`
	`  }`
	`  Report.Stderr = writer;`
	`  return result;`
	`}`

Example Notes:	The example method begins by saving the current number of warnings, and the error reporting stream. The method then, sets the error reporting stream to null. The purpose of this is to suppress the reporting of the warning to the user. After calling the *CheckObsoleteness* method, if the number of warnings has changed, then an "Obsolete" attribute has been found. The example method finishes, by resetting the error reporting object, and returning the result.
Thread Safe:	No
C# Method Signature:	public virtual void CheckObsoleteness(Location loc);
Version Information:	New in Version 0.2.5.1 of the SDK

GetClassBases

<table>
<tr><td>Name:</td><td>GetClassBases</td></tr>
<tr><td>Description:</td><td>Computes and returns the base class and the list of interfaces that the type implements</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Out TypeExpr</td><td>The base type the class or structure extends</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The GetClassBases method computes the base class and also the list of interfaces that the class or structure implements. The return value is an array (might be null) of interfaces implemented (as types). The base class parameter is set to the base object or null if this is “System.Object”.</td></tr>
<tr><td>C# Example:</td><td>
<pre>
public static void TestBases(TypeContainer parent)
{
    TypeExpr baseClass;

    TypeExpr[] interfaces = parent.GetClassBases(out baseClass);
    if (baseClass != null)
        ProcessType(baseClass);
    if (interfaces != null)
        foreach (TypeExpr te in interfaces)
            ProcessType(te);
}
</pre>
</td></tr>
<tr><td>Example Notes:</td><td>Gets any bases for the given type, and calls the “ProcessType” method for each base referenced by the given type.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public TypeExpr[] GetClassBases(out TypeExpr base_class);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

GetClsCompliantAttributeValue

Name:	**GetClsCompliantAttributeValue**
Description:	**Returns true if the containing type has the ClsCompliant attribute applied to it**
Parameters:	0
Parameter Values:	N/A
Remarks:	The *GetClsCompliantAttributeValue* method returns true if the containing type contains the ClsCompliant attribute. If the attribute is not found, the *GetClsCompliantAttributeValue* method then examines the entire class hierarchy to search for the attribute, if the attribute is still not found, the assembly wide attributes are examined. The method will return true if the search yields a ClsCompliant attribute. If such an attribute cannot be found, then the method will return false. If you are writing .NET classes, which will be used by other .NET classes irrespective of the language they are implemented, then your code should conform to the CLS (Common Language Specification). This means that your class should only expose features that are common across all .NET languages. The following are the basic rules that should be followed when writing CLS complaint C# code. 1. Unsigned types should not be part of the public interface of the class. What this means is public fields should not have unsigned types like uint or ulong, public methods should not return unsigned types, parameters passed to public function should not have unsigned types. However unsigned types can be part of private members. 2. Unsafe types like pointers should not be used with public members. However they can be used with private members. 3. Class names and member names should not differ only based on their case. For example you cannot have two methods named MyMethod and MYMETHOD. 4. Only properties and methods may be overloaded. Operators should not be overloaded. The ClsCompliant attribute is used to indicate if a program element is compliant with the Common Language Specification (CLS). If the ClsCompliant attribute is not applied then the following holds: 1. The assembly is not CLS Compliant 2. The type is CLS Compliant only if it's enclosing type or assembly is CLS Compliant 3. The member of a type is CLS Compliant only if the type is CLS Compliant

C# Example:	public static bool IsClsCompliant(TypeContainer parent)
	{
	return parent.GetClsCompliantAttributeValue();
	}

Example Notes:	The example method returns true if the given type container's class hiearchy contains a ClsCompliant attribute that is set to true. If such an attribute cannot be located, or the attribute value is set to false, the example method will also return false.
Thread Safe:	No
C# Method Signature:	public bool GetClsCompliantAttributeValue();
Version Information:	New in Version 0.2.5.1 of the SDK

GetDefinition

Name:	**GetDefinition**
Description:	**Returns the MemberCore associated with the given name in the type container**
Parameters:	1
Parameter Values:	Ordinal: 1 — Type: String — Description: The name to search for
Remarks:	The *GetDefinition* method searches the current type for a member that is associated with the given name. That is, the member that is identified by the given name is returned.

C# Example:	`public static void ProcessMemberByName(TypeContainer parent,`
	`  string name)`
	`{`
	`  MemberCore core = parent.GetDefinition(name);`
	`  if (core != null)`
	`  ProcessMember(core);`
	`}`

Example Notes:	The example method gets the member defined by the given name. Then if the resultant member is valid the "ProcessMember" method is called to process the actual member.
Thread Safe:	No
C# Method Signature:	`public MemberCore GetDefinition(string name);`
Version Information:	New in Version 0.2.5.1 of the SDK

GetDocCommentName

<table>
<tr><td>Name:</td><td>GetDocCommentName</td></tr>
<tr><td>Description:</td><td>Returns a string that represents the signature for the containing type</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>DeclSpace</td><td>The containing declaration space</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The GetDocCommentName method returns a string that represents the signature for the given type. This string should be used for generating XML documentation or when writing information to the Contract Evaluation Report.

The “DeclSpace” parameter specifies the containing declaration space, if any. If there is no containing declaration space, then a value of “null” should be used.</td></tr>
<tr><td>C# Example:</td><td>public static string GetDocName(TypeContainer parent)
{
    return parent.GetDocCommentName(null);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method gets the document comment name for the given type container. There is no containing declaration space.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public virtual string GetDocCommentName(DeclSpace ds);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

GetMethods

Name:	**GetMethods**
Description:	**Gets a MethodInfo instance for every method defined within the type**
Parameters:	o
Parameter Values:	N/A
Remarks:	The *GetMethods* method is used to get a System.Reflection.MethodInfo instance for every method defined within the type. The *GetMethods* method returns an array of type MethodInfo. The array will be null of the type does not define any methods. The *GetMethods* method should not be used to retrieve method information, rather the *Methods* property should be used to access method information, especially when doing analysis of method bodies. The *GetMethods* method is included for developers who wish to modify or analyze the MethodInfo information directly.

C# Example:	public static void TestMethodInfos(TypeContainer parent)
	{
	MethodInfo[] methods = parent.GetMethods();
	if (methods != null)
	foreach (MethodInfo m in methods)
	TestMethodInfo(m);
	}

Example Notes:	Gets a MethodInfo instance for each method defined in the given type, and then calls the "TestMethodInfo" method on each one that was returned.
Thread Safe:	No
C# Method Signature:	public MethodInfo[] GetMethods();
Version Information:	New in Version 0.2.5.1 of the SDK

GetObsoleteAttribute

Name:	**GetObsoleteAttribute**
Description:	**Gets an instance of the obsolete attribute for this type**
Parameters:	0
Parameter Values:	N/A
Remarks:	The *GetObsoleteAttribute* method returns an instance of the obsolete attribute if one is applied to the containing type. If no such attribute is applied the *GetObsoleteAttribute* method returns null. Obsolete program elements will issue a compile time warning when they are referenced in code. The idea is to mark a given program element as obsolete to discourage its usage and to indicate to the user that such element should no longer be used. The obsolete attribute, is defined by the "ObsoleteAttribute" class. This class is part of the .NET Framework SDK. For more information please see the SDK documentation [7].

C# Example:	public static bool IsObsolete(TypeContainer parent)
	{
	ObsoleteAttribute attr = parent.GetObsoleteAttribute();
	return (attr != null);
	}

Example Notes:	The example method uses the *GetObsoleteAttribute* method to get the "ObsoleteAttribute" instance if availiable. The example method will return true if the given type container is marked with the Obsolete attribute.
Thread Safe:	No
C# Method Signature:	public virtual ObsoleteAttribute GetObsoleteAttribute();
Version Information:	New in Version 0.2.5.1 of the SDK

GetSignatureForError

Name:	**GetSignatureForError**
Description:	**Gets a string representation of the type container's signature**
Parameters:	o
Parameter Values:	N/A
Remarks:	The *GetSignatureForError* method returns a string representation of the type container's signature. This signature can be used for generating errors which reference the type container. The method can also be used for generating messages to return to the contract evaluation engine regarding the type container. This method is included for developer assistance. The *GetSignatureForError* method does not have any functional purpose, and does not modify the type container in any way.
C# Example:	`public static bool VerifyClsCompliance(TypeContainer parent)` `{` `  parent.VerifyClsName();` `  Type base_type = parent.TypeBuilder.BaseType;` `  if (base_type != null &&` `  !AttributeTester.IsClsCompliant(base_type))` `  {` `  Report.Error(3009, parent.Location,` `  "`{0}': base type `{1}' is not CLS-compliant",` `  parent.GetSignatureForError(),` `  TypeManager.CSharpName(base_type));` `  }` `  if (!parent.Parent.IsClsComplianceRequired(parent))` `  {` `  Report.Error(3018, parent.Location,` `  "`{0}' cannot be marked as CLS-compliant" +` ` "because it is a member of non CLS-compliant" +` ` "type `{1}'", parent.GetSignatureForError(),` `  parent.Parent.GetSignatureForError());` `  }` `  return true;` `}`
Example Notes:	The example method above checks to see if the given type container is CLS Compliant. If an error is encountered the *GetSignatureForError* method is used to get a string representation of the type container, and in some cases, the parent's base type name.
Thread Safe:	Yes
C# Method Signature:	`public override string GetSignatureForError();`
Version Information:	New in Version 0.2.5.1 of the SDK

LookupType

<table>
<tr><td>Name:</td><td>LookupType</td></tr>
<tr><td>Description:</td><td>Returns the type specified by the given name</td></tr>
<tr><td>Parameters:</td><td>3</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>String</td><td>The type name to search for</td></tr>
<tr><td>2</td><td>Location</td><td>The source code location where the lookup is taking place (or is required)</td></tr>
<tr><td>3</td><td>Boolean</td><td>True if we are to ignore C# error 104</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The LookupType method is used to located types. The current type container is used to define scope. That is, only types that are visible from the containing type container will be examined. The first parameter specifies the name of the type that is requested.

The second parameter is a location object. The location is used when reporting type lookup errors so that the compiler can generate a useful error message. When calling the LookupType method in a static extension, the current type container's "Location" property can be used to specify the location used for the LookupType method call.

The third parameter specifies if C# compiler error 104 should be reported or not. C# compiler error 104 (CS0104) is issued when there is an ambiguous reference between two identifieres. That is, the compiler cannot determine which reference to return. If the error is supressed, the first matching type will be returned. When calling LookupType in a static extension the third parameter should always be set to true, because any compiler errors will be given to the user in the Contract Editor's Log output window.

If a matching type can be found, the LookupType method returns an instance of the FullNamedExpression class. A FullNamedExpression is simply a specialized expression which can be used in method bodies to reference the given type. For more information on method bodies, please see the discussion on the Method parameter type later in this document. If no matching type can be found, then LookupType will return null.</td></tr>
</table>

C# Example:	public static Expression GetExpressionForType(TypeContainer parent,
	string name)
	{
	return parent.LookupType(name, parent.Location, true);
	}

Example Notes:	The example method will return a valid FullNamedExpression if the type denoted by the given string, is visible from the given type container. If no such type can be located, the method will return null. The given type container's source code location is used as the reference location, and the CS0104 error is suppressed.
Thread Safe:	No
C# Method Signature:	public FullNamedExpression LookupType(string name, Location loc, bool ignore_cs0104);
Version Information:	New in Version 0.2.5.1 of the SDK

IsClsComplianceRequired

<table>
<tr><td>Name:</td><td>IsClsComplianceRequired</td></tr>
<tr><td>Description:</td><td>Checks to see if this type container must be CLS Compliant</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>DeclSpace</td><td>The containing type if any</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>
The IsClsComplianceRequired method performs checks to see if the entire type container is required to be CLS Compliant or not.

The “DeclSpace” parameter is used to specify the containing type which contains the type container, if any. If the type container does not have a containing type, a value of null should be used.

If you are writing .NET classes, which will be used by other .NET classes irrespective of the language they are implemented, then your code should conform to the CLS (Common Language Specification). This means that your class should only expose features that are common across all .NET languages. The following are the basic rules that should be followed when writing CLS complaint C# code.

1. Unsigned types should not be part of the public interface of the class. What this means is public fields should not have unsigned types like uint or ulong, public methods should not return unsigned types, parameters passed to public function should not have unsigned types. However unsigned types can be part of private members.
2. Unsafe types like pointers should not be used with public members. However they can be used with private members.
3. Class names and member names should not differ only based on their case. For example you cannot have two methods named MyMethod and MYMETHOD.
4. Only properties and methods may be overloaded. Operators should not be overloaded.

The IsClsComplianceRequired will return true if the type container must be CLS Compliant, false otherwise.

For more information on CLS Compliance please see the .NET Framework 2.0 SDK documentation [7].
</td></tr>
</table>

C# Example:	public static bool MustBeClsCompliant(TypeContainer parent)
	{
	return parent.IsClsComplianceRequired(null);
	}

Example Notes:	The example method uses the *IsClsComplianceRequired* method to determine if the given type container is required to be CLS Compliant or not.
Thread Safe:	No
C# Method Signature:	public override bool IsClsComplianceRequired(DeclSpace container);
Version Information:	New in Version 0.2.5.1 of the SDK

IsExposedFromAssembly

Name:	**IsExposedFromAssembly**
Description:	**Checks to see if this type container is exposed from the assembly**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>DeclSpace</td><td>The containing type if any</td></tr></table>
Remarks:	The *IsExposedFromAssembly* method performs checks to see if the entire type container is exposed from the assembly or not. Being exposed from an assembly simply means that program elements that reside outside of the type container's assembly are able to view the type container. That is, the type container's accessibility modifiers allow it to be seen outside of the assembly. The "DeclSpace" parameter is used to specify the containing type which contains the type container, if any. If the type container does not have a containing type, a value of null should be used. The *IsExposedFromAssembly* will return true if the type container is visible from outside the assembly, false otherwise.

C# Example:	public static bool IsTypeContainerExposed(TypeContainer parent)
	{
	return parent.IsExposedFromAssembly(null);
	}

Example Notes:	The example method uses the *IsTypeContainerExposed* method to determine if the given type container is visible from outside the assembly or not.
Thread Safe:	No
C# Method Signature:	public bool IsExposedFromAssembly(DeclSpace ds);
Version Information:	New in Version 0.2.5.1 of the SDK

SetMemberIsUsed

Name:	**SetMemberIsUsed**
Description:	**Marks the entire type container as being used**
Parameters:	o
Parameter Values:	N/A
Remarks:	The *SetMemberIsUsed* method sets an internal flag in the type container to mark it as used within the current code base. Normally, a program element's usage is calculated automatically by the C# compiler. However, sometimes when adding code via a static extension, the compiler may indicate that a given program element is not used. This method will allow you to remove that warning from the compiler output.

C# Example:	public static void MarkAsUsed(TypeContainer parent)
	{
	parent.SetMemberIsUsed();
	}

Example Notes:	The example method uses the *SetMemberIsUsed* method to mark the given type container as used.
Thread Safe:	Yes
C# Method Signature:	public void SetMemberIsUsed();
Version Information:	New in Version 0.2.5.1 of the SDK

UnsafeOK

<table>
<tr><td>Name:</td><td>UnsafeOK</td></tr>
<tr><td>Description:</td><td>Checks to see if this type container can use pointers</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>DeclSpace</td><td>The containing type if any</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The UnsafeOK method performs checks to see if it is okay to use an unsafe pointer within the type container.

The “DeclSpace” parameter is used to specify the containing type which contains the type container, if any. If the type container does not have a containing type, a value of null should be used.

The UnsafeOK method will return true if the type container can use unsafe pointers, false otherwise.</td></tr>
<tr><td>C# Example:</td><td>public static bool CanUsePointers(TypeContainer parent)
{
    return parent.UnsafeOK(parent.Parent);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method uses the UnsafeOK method to determine if the given type container can use unsafe pointers or not.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public bool UnsafeOK(DeclSpace parent);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

VerifyImplements

Name:	**VerifyImplements**
Description:	**Checks for an explicit interface implementation**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>MemberBase</td><td>The member to check for</td></tr></table>
Remarks:	The *VerifyImplements* method performs checks for an explicit interface implementation using the given member. First it checks whether the parameter's type is a base interface implementation, and then it checks to see if the name matches. If the both checks succeed then *VerifyImplements* returns true, the method returns false otherwise. The MemberBase class is used to derive the Constructor, Field, Indexer, Method, Property, and Operator classes, so the *VerifyImplements* method is able to check for any element type.

C# Example:	public static bool TestExplicitInterfaceImpl(
	TypeContainer parent, MemberBase base)
	{
	return parent.VerifyImplements(base);
	}

Example Notes:	Uses the *VerifyImplements* method to determine if the given type explicitily implements the given member.
Thread Safe:	No
C# Method Signature:	public bool VerifyImplements(MemberBase base);
Version Information:	New in Version 0.2.5.1 of the SDK

Properties

The following tables will discuss each of the properties defined by the TypeContainer type. These properties can be used to access the structural elements contained within the TypeContainer.

AttributeTargets

Name:	**AttributeTargets**
Description:	**Gets the type of attributes which can be applied to the type container**
Read Only:	Yes
C# Signature:	public AttributeTargets AttributeTargets
Remarks:	The *AttributeTargets* property returns a System.AttributeTargets enumeration value indicating the type of attributes which can be applied to this type container. The result is based on the kind of type the type container represents. The possible return values are as follows: AttributeTargets.Class; AttributeTargets.Struct; AttributeTargets.Interface;

C# Example:	public static AttributeTargets GetAttributeTargets(
	TypeContainer parent)
	{
	return parent.AttributeTargets;
	}

Example Notes:	Uses the property to get the type of attributes which can be applied to the given type container.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Base

Name:	**Base**
Description:	**Gets the base class name**
Read Only:	Yes
C# Signature:	public string Base
Remarks:	The *Base* property returns a string representation of the base class. If there is no base class an empty string will be returned.

C# Example:	public static bool IsBase(TypeContainer parent, string name)
	{
	return (parent.Base == name);
	}

Example Notes:	Uses the property to check if the base class name of the given type matches the given name.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

BaseCache

Name:	**BaseCache**
Description:	**Gets the base member cache for lookups**
Read Only:	Yes
C# Signature:	public virtual MemberCache BaseCache
Remarks:	The *BaseCache* property returns the member cache for the base type. Member caches are used for looking up members for identifier resolution. The MemberCache type is used by dynamic and non-dynamic types to speed up member lookups. A dynamic type is one that is being currently defined by the SUT, where a non-dynamic type is one that has already been defined in some external library but is being used by the SUT. The cache has a member name based hash table; it maps each member name to a list of CacheEntry objects. Each CacheEntry contains a MemberInfo and the BindingFlags that were initially used to get it. The cache contains all members of the current class and all inherited members. If this cache is for an interface type, it also contains all inherited members. There are two ways to get a MemberCache: 1) If this is a dynamic type, lookup the corresponding DeclSpace and then use the DeclSpace.MemberCache property. TypeContainer subclasses from DeclSpace. 2) If this not a dynamic type, call TypeHandle.GetTypeHandle() to get a TypeHandle instance for the type and then use TypeHandle.MemberCache.

C# Example:	public static MethodInfo FindOverriddenMethod(
	TypeContainer parent, string name, Type[] parameterTypes)
	{
	MethodInfo mi =
	(MethodInfo)parent.BaseCache.FindMemberToOverride(
	parent.TypeBuilder, name, parameterTypes, false);
	return mi;
	}

Example Notes:	Uses the property to get the member cache. The member cache is used to find a MethodInfo which is overridden by the given type container. The method to search for is defined using the given name and parameter types. If no method is found the example code will return null.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Bases

Name:	**Bases**
Description:	**Gets all the bases which are inherited/implemented by this type**
Read Only:	No
C# Signature:	public ArrayList Bases
Remarks:	The *Bases* property returns an array list of all of the bases which are inherited or implemented by this type. Each element of the array list is an instance of the TypeExpr class. The TypeExpr class represents a type expression and consists of a Name property, and other Boolean properties such as IsClass, IsInterface, and IsValueType. In the resultant array of TypeExprs, if there is a base type it will be located at index zero, all interfaces will follow.

C# Example:	public static TypeExpr[] GetBases(TypeContainer parent,
	out TypeExpr base_class)
	{
	base_class = null;
	if (parent.Bases == null) return null;
	if (parent.Bases.Count == 0) return null;
	
	TypeExpr[] ifaces;
	int start = 0;
	if ((parent.Bases[0] as TypeExpr).IsClass)
	{
	base_class = parent.Bases[0] as TypeExpr;
	ifaces = new TypeExpr[parent.Bases.Count - 1];
	++start;
	}
	else
	ifaces = new TypeExpr[parent.Bases.Count];
	
	for (int i = start; i < parent.Bases.Count; i++)
	ifaces[i] = (TypeExpr)parent.Bases[i];
	
	return ifaces;
	}

Example Notes:	Uses the property to get a list of all bases, then sets the base_class variable to the base class, if any. The example code then calculates and returns a list of any interfaces implemented by the given type.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Constants

Name:	**Constants**
Description:	**Gets all the constants which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Constants
Remarks:	The *Constants* property returns an array list of all of the constants which are contained within this type. Each element of the array list is an instance of the Const class. The Const class represents a single constant member.

C# Example:	public static void ProcessConstants(TypeContainer parent)
	{
	if (parent.Constants != null)
	foreach (Const c in parent.Constants)
	ProcessConstant(c);
	}

Example Notes:	The example code iterates through each of the constants contained within the given type and calls the "ProcessConstant" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

DefaultStaticConstructor

Name:	**DefaultStaticConstructor**
Description:	**Gets the default static constructor for the type container**
Read Only:	Yes
C# Signature:	public Constructor DefaultStaticConstructor
Remarks:	The *DefaultStaticConstructor* property returns the default static constructor for the type container. If such a constructor does not exist, a value of null will be returned. All constructors are represented by the Constructor class, which is a subclass of MethodCore. That is, a constructor contains a method body which can be analyzed for behavioural analysis. Details of such analysis will be presented later in this document in the discussion on the Method parameter type.
C# Example:	public static bool HasUserDefinedStaticConstructor(TypeContainer parent) { return (parent.DefaultStaticConstructor != null); }
Example Notes:	The example code returns true if the given type container has a static constructor defined.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Delegates

Name:	**Delegates**
Description:	**Gets all the delegates which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Delegates
Remarks:	The *Delegates* property returns an array list of all of the delegates which are contained within this type. Each element of the array list is an instance of the Delegate class. The Delegate class represents a single delegate member.

C# Example:	public static void ProcessDelegates(TypeContainer parent)
	{
	if (parent.Delegates != null)
	foreach (Delegate d in parent.Delegates)
	ProcessDelegate(d);
	}

Example Notes:	The example code iterates through each of the delegates contained within the given type and calls the "ProcessDelegate" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

DocCommentHeader

Name:	**DocCommentHeader**
Description:	**Gets the xml document header string for xml comment output**
Read Only:	Yes
C# Signature:	public override string DocCommentHeader
Remarks:	The *DocCommentHeader* property returns the xml document header string for xml comment output. This property is only used for xml comment generation. It is included in the Contract Evaluation Engine Extension SDK so that if xml code comments are being generated as part of static evaluation the necessary API is available.

C# Example:	public static string GetDocCommentName(TypeContainer parent)
	{
	return String.Concat(parent.DocCommentHeader,
	parent.Name);
	}

Example Notes:	The example code returns the correct xml comment name for the given type container.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Enums

Name:	**Enums**
Description:	**Gets all the enumerations which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Enums
Remarks:	The *Enums* property returns an array list of all of the enumerations which are contained within this type. Each element of the array list is an instance of the Enum class. The Enum class represents a single enumeration member.

C# Example:	public static void ProcessEnumerations(TypeContainer parent)
	{
	if (parent.Enums != null)
	foreach (Enum e in parent.Enums)
	ProcessEnumeration(e);
	}

Example Notes:	The example code iterates through each of the enumerations contained within the given type and calls the "ProcessEnumeration" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Events

Name:	**Events**
Description:	**Gets all the events which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Events
Remarks:	The *Events* property returns an array list of all of the events which are contained within this type. Each element of the array list is an instance of the Event class. The Event class represents a single event member.

C# Example:	public static void ProcessEvents(TypeContainer parent)
	{
	if (parent.Events != null)
	foreach (Event e in parent.Events)
	ProcessEvent(e);
	}

Example Notes:	The example code iterates through each of the events contained within the given type and calls the "ProcessEvent" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Fields

Name:	**Fields**
Description:	**Gets all the fields which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Fields
Remarks:	The *Fields* property returns an array list of all of the fields which are contained within this type. Each element of the array list is an instance of the Field class. The Field class represents a single field member.

C# Example:	public static void ProcessFields(TypeContainer parent)
	{
	if (parent.Fields != null)
	foreach (Field f in parent.Fields)
	ProcessField(f);
	}

Example Notes:	The example code iterates through each of the fields contained within the given type and calls the "ProcessField" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

HasExplicitLayout

Name:	**HasExplicitLayout**
Description:	**Indicates whether the type container has an explicit layout**
Read Only:	Yes
C# Signature:	public virtual bool HasExplicitLayout

Remarks:	The *HasExplicitLayout* property indicates whether the type container has an explicit layout. An explicit layout is given, when the type container has the "StructLayout" attribute applied, and it is set to "Explicit".
	
	Explicit layouts are needed, when creating structures for use to transfer data from unmanaged APIs to managed ones. For example, the WIN32 Rect structure can be defined in managed code as follows:
	
	[StructLayout(LayoutKind.Explicit)]
	public struct Rect
	{
	[FieldOffset(0)] public int left;
	[FieldOffset(4)] public int top;
	[FieldOffset(8)] public int right;
	[FieldOffset(12)] public int bottom;
	}

C# Example:	public static bool HasExplicitLayout(TypeContainer parent)
	{
	return parent.HasExplicitLayout;
	}

Example Notes:	The example code returns true if the given type container has an explicit layout, false otherwise.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

IndexerName

Name:	**IndexerName**
Description:	**Gets the name of the indexer for this type**
Read Only:	Yes
C# Signature:	public string IndexerName
Remarks:	The *IndexerName* property returns a string representation for the name of the indexer defined on this type. If no indexer is defined on this type, the default name of "Item" will be returned.

C# Example:	public static bool HasDefaultIndexer(TypeContainer parent)
	{
	return (parent.IndexerName == "Item");
	}

Example Notes:	The example code returns true if the given type container uses the default indexer. That is, the type does not define a specialized indexer.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Indexers

Name:	**Indexers**
Description:	**Gets all the indexers which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Indexers
Remarks:	The *Indexers* property returns an array list of all of the indexers which are contained within this type. Each element of the array list is an instance of the Indexer class. The Indexer class represents a single indexer member.

C# Example:	public static void ProcessIndexers(TypeContainer parent)
	{
	if (parent.Indexers != null)
	foreach (Indexer i in parent.Indexers)
	ProcessIndexer(i);
	}

Example Notes:	The example code iterates through each of the indexers contained within the given type and calls the "ProcessIndexer" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

InstanceConstructors

Name:	**InstanceConstructors**
Description:	**Gets all the instance constructors which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList InstanceConstructors
Remarks:	The *InstanceConstructors* property returns an array list of all of the instance (non-static) constructors which are contained within this type. Each element of the array list is an instance of the Constructor class. The Constructor class represents a single constructor member, which is a subclass of MethodCore. That is, a constructor contains a method body which can be analyzed for behavioural analysis. Details of such analysis will be presented later in this document in the discussion on the Method parameter type.

C# Example:	public static void ProcessInstanceConstructors(TypeContainer parent)
	{
	if (parent.InstanceConstructors != null)
	foreach (Constructor c in parent.InstanceConstructors)
	ProcessConstructor(c);
	}

Example Notes:	The example code iterates through each of the instance constructors contained within the given type and calls the "ProcessConstructor" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Interfaces

Name:	**Interfaces**
Description:	**Gets all the interfaces which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Interfaces
Remarks:	The *Interfaces* property returns an array list of all of the interfaces which are contained within this type. Each element of the array list is an instance of the Interface class. The Interface class represents a single constructor member, which is a subclass of TypeContainer. That is, an interface contains all of the methods and properties discussed in this section, and they can be used to analyze/modify the interface as needed.

C# Example:	public static void ProcessInterfaces(TypeContainer parent)
	{
	if (parent.Interfaces != null)
	foreach (Interface i in parent.Interfaces)
	ProcessInterface(i);
	}

Example Notes:	The example code iterates through each of the interfaces contained within the given type and calls the "ProcessInterface" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

InUnsafe

Name:	**InUnsafe**
Description:	**Determines if this type container is marked as unsafe**
Read Only:	Yes
C# Signature:	public bool InUnsafe
Remarks:	The *InUnsafe* property indicates if the given type container is marked as unsafe. Unsafe types are able to use pointers. A type container is marked as unsafe by using the "unsafe" keyword as a modifier when the type container is defined.

C# Example:	public static bool IsTypeContainerUnsafe(TypeContainer parent)
	{
	return parent.InUnsafe;
	}

Example Notes:	The example code uses the *InUnsafe* property to determine if the given type container is marked as unsafe or not.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

IsComImport

Name:	**IsComImport**
Description:	**Determines if this type container contains the ComImport attribute**
Read Only:	Yes
C# Signature:	public bool IsComImport

Remarks:	The *IsComImport* property returns true of the type has the ComImport attribute assigned to it, false otherwise.
	
	The ComImport attribute is a pseudo-custom attribute that indicates that a type has been defined in a previously published type library. The common language runtime treats these types differently when activating, exporting, coercing, and so on. The following is an example of the ComImport attribute:
	
	[ComImport]
	[Guid("73EB4AF8-BE9C-4b49-B3A4-24F4FF657B26")]
	public interface IMyStorage
	{
	[DispId(1)]
	[return: MarshalAs(UnmanagedType.Interface)]
	Object GetItem([In, MarshalAs(UnmanagedType.BStr)]
	String bstrName);
	
	[DispId(2)]
	void GetItems([In, MarshalAs(UnmanagedType.BStr)]
	String bstrLocation,
	[Out, MarshalAs(UnmanagedType.SafeArray,
	SafeArraySubType = VarEnum.VT_VARIANT)]
	out Object[] Items);
	
	[DispId(3)]
	void GetItemDescriptions([In] String bstrLocation,
	[In, Out, MarshalAs(UnmanagedType.SafeArray)]
	ref Object[] varDescriptions);
	
	bool IsEmpty
	{
	[DispId(4)]
	[return: MarshalAs(UnmanagedType.VariantBool)]
	get;
	}
	}

C# Example:	public static bool IsComImport(TypeContainer parent)
	{
	  return parent.IsComImport;
	}

Example Notes:	The example code returns true if the given type contains the ComImport attribute, false otherwise.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

IsTopLevel

Name:	**IsTopLevel**
Description:	**Determines if this type container is a top level type**
Read Only:	Yes
C# Signature:	public bool IsTopLevel
Remarks:	The *IsTopLevel* property indicates if the given type is a top level type or not. If the type container is a top level type then the property will return true. Otherwise, the *IsTopLevel* property will return false.

C# Example:	public static bool IsTypeTopLevel(TypeContainer parent)
	{
	  return parent.IsTopLevel;
	}

Example Notes:	The example code uses the *IsTopLevel* property to determine if the given type container is a top level type or not.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

IsUsed

Name:	**IsUsed**
Description:	**Determines if this type container is used or not**
Read Only:	Yes
C# Signature:	public virtual bool IsUsed
Remarks:	The *IsUsed* property indicates if the given type container is used or referenced by another program element. The property will return true if the type container is used by another program element, false otherwise. Developers can mark a type container as used via the "SetMemberIsUsed" method. The method has been previously discussed.

C# Example:	public static bool IsTypeContainerUsed(TypeContainer parent)
	{
	return parent.IsUsed;
	}

Example Notes:	The example code uses the *IsUsed* property to determine if the given type container is used or referenced by another program element.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Iterators

Name:	**Iterators**
Description:	**Gets all the iterators which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Iterators
Remarks:	The *Iterators* property returns an array list of all of the iterators which are contained within this type. Each element of the array list is an instance of the Iterator class. The Iterator class represents a single iterator member, which is a subclass of TypeContainer. That is, an iterator contains all of the methods and properties discussed in this section, and they can be used to analyze/modify the iterator as needed.

C# Example:	public static void ProcessIterators(TypeContainer parent)
	{
	if (parent.Iterators != null)
	foreach (Iterator i in parent.Iterators)
	ProcessIterator(i);
	}

Example Notes:	The example code iterates through each of the iterators contained within the given type and calls the "ProcessIterator" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Location

Name:	**Location**
Description:	**Gets source code location where the type container was defined**
Read Only:	Yes
C# Signature:	public Location Location
Remarks:	The *Location* property represents the source location where the type container was defined. That is, the source code file name and line number. The line number specified is where the first line of the type container's definition is located. The "Location" class contains several members to access the source code file, line number, and column where the type container's definition is located.

C# Example:	public static int GetTypeContainerDefinitionLineNumber(
	TypeContainer parent)
	{
	return parent.Location.Row;
	}

Example Notes:	The example code uses the *Location* property to access the source code location where the type container was defined. Once the location information is retrieved, the example code returns the line number (row) where the definition is located.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

MemberCache

Name:	**MemberCache**
Description:	**Gets the member cache for lookups**
Read Only:	Yes
C# Signature:	public override MemberCache MemberCache
Remarks:	The *MemberCache* property returns the member cache for the type container. Member caches are used for looking up members for identifier resolution. The MemberCache type is used by dynamic and non-dynamic types to speed up member lookups. It has a member name based hash table; it maps each member name to a list of CacheEntry objects. Each CacheEntry contains a MemberInfo and the BindingFlags that were initially used to get it. The cache contains all members of the current class and all inherited members. If this cache is for an interface type, it also contains all inherited members. There are two ways to get a MemberCache: 1) If this is a dynamic type, lookup the corresponding DeclSpace and then use the DeclSpace.MemberCache property. TypeContainer subclasses from DeclSpace. 2) If this not a dynamic type, call TypeHandle.GetTypeHandle() to get a TypeHandle instance for the type and then use TypeHandle.MemberCache.

C# Example:	public static MethodInfo FindOverriddenMethod(
	TypeContainer parent, string name, Type[] parameterTypes)
	{
	MethodInfo mi =
	(MethodInfo)parent.MemberCache.FindMemberToOverride(
	parent.TypeBuilder, name, parameterTypes, false);
	return mi;
	}

Example Notes:	The example uses the *MemberCache* property to get the member cache. The member cache is used to find a MethodInfo which is overridden by the given type container. The method to search for is defined using the given name and parameter types. If no method is found the example code will return null.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

MemberName

Name:	**MemberName**
Description:	**Gets the member name object that represents the type container**
Read Only:	Yes
C# Signature:	public MemberName MemberName
Remarks:	The *MemberName* property returns a member name object which represents the name of the containing type container. The "MemberName" type contains the name, namespace, and source code location where the type container has been defined. Each of these elements can be accessed via dedicated properties defined on the "MemberName" type. If you are only interested in the name of the type container, use the "Name" property instead.

C# Example:	public static string GetFullName(TypeContainer parent)
	{
	return parent.MemberName.GetName();
	}

Example Notes:	The example code uses the *GetMemberName* property to get the member name object, and then calls the "GetName" method defined on the "MemberName" type. The "GetName" method returns a string representation of the fully resolved name of the given type container.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Methods

Name:	**Methods**
Description:	**Gets all the methods which are contained within this type**
Read Only:	Yes
C# Signature:	public MethodArrayList Methods
Remarks:	The *Methods* property returns an array list of all of the methods which are contained within this type. Each element of the array list is an instance of the Method class. The Method class represents a single method member, which is a subclass of MethodCore. That is, a method contains a method body which can be analyzed for behavioural analysis. Details of such analysis will be presented later in this document in the discussion on the Method parameter type.

C# Example:	public static void ProcessMethods(TypeContainer parent)
	{
	if (parent.Methods != null)
	foreach (Method m in parent.Methods)
	ProcessMethod(m);
	}

Example Notes:	The example code iterates through each of the methods contained within the given type and calls the "ProcessMethod" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Name

Name:	**Name**
Description:	**Gets the name of this type container**
Read Only:	Yes
C# Signature:	public string Name
Remarks:	The *Name* property returns a string representation of the type container's name. This name will include namespace information as well as the type container's actual name. The *Name* property yields the same result as calling "MemberName.GetName(false)" however, the name is cached so the *Name* property should be used instead.

C# Example:	public static string GetTypeContainerName(TypeContainer parent)
	{
	return parent.Name;
	}

Example Notes:	The example code uses the *Name* property to get the full name of the given type container.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Operators

Name:	**Operators**
Description:	**Gets all the operators which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Operators
Remarks:	The *Operators* property returns an array list of all of the operators which are contained within this type. Each element of the array list is an instance of the Operator class. The Operator class represents a single operator member, which is a subclass of MethodCore. That is, an operator contains a method body which can be analyzed for behavioural analysis. Details of such analysis will be presented later in this document in the discussion on the Method parameter type.

C# Example:	public static void ProcessOperators(TypeContainer parent)
	{
	if (parent.Operators != null)
	foreach (Operator o in parent.Operators)
	ProcessOperator(o);
	}

Example Notes:	The example code iterates through each of the operators contained within the given type and calls the "ProcessOperator" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

OptAttributes

Name:	**OptAttributes**
Description:	**Contains a list of all the attributes which have been applied to this type container**
Read Only:	No
C# Signature:	public Attributes OptAttributes
Remarks:	The *OptAttributes* property contains a list of all the attributes which have been applied to this type container. The *OptAttributes* property also allows for new attributes to be applied programmatically.

C# Example:	public static void AddAttribute(TypeContainer parent,
	Attribute newAttribute)
	{
	if (parent.OptAttributes == null)
	parent.OptAttributes = new Attributes(newAttribute);
	else
	parent.OptAttributes.Attrs.Add(newAttribute);
	}

Example Notes:	The example code adds a new attribute to the given type container. There are two cases. The first case, is where the new attribute is the first attribute to be added to the type container. In that case, a new "Attributes" container is created and the new attribute is added via the construction. The second case, is where the new attribute is being added to the existing list of attributes. This is accomplished via the "Add" method, defined on the actual attribute list. The actual attribute list is exposed via the "Attrs" property.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Parts

Name:	**Parts**
Description:	**Gets all the partial type definitions which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Parts
Remarks:	The *Parts* property returns an array list of all of the partial type definitions which are contained within this type. Each element of the array list is an instance of the ClassPart class. The ClassPart class represents a single partial class member, which is a subclass of TypeContainer. That is, a class part contains all of the methods and properties discussed in this section, and they can be used to analyze/modify the class part as needed.

C# Example:	public static void ProcessParts(TypeContainer parent)
	{
	if (parent.Parts != null)
	foreach (ClassPart cp in parent.Parts)
	ProcessPart(cp);
	}

Example Notes:	The example code iterates through each of the partial type definitions contained within the given type and calls the "ProcessPart" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Properties

Name:	**Properties**
Description:	**Gets all the properties which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Properties
Remarks:	The *Properties* property returns an array list of all of the properties which are contained within this type. Each element of the array list is an instance of the Property class. The Property class represents a single property member. The property member contains up to two accessors, which are subclasses of MethodCore. That is, an accessor contains a method body which can be analyzed for behavioural analysis. Details of such analysis will be presented later in this document in the discussion on the Method parameter type.

C# Example:	public static void ProcessProperties(TypeContainer parent)
	{
	if (parent.Properties != null)
	foreach (Property p in parent.Properties)
	ProcessProperty(p);
	}

Example Notes:	The example code iterates through each of the properties contained within the given type and calls the "ProcessProperty" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Types

Name:	**Types**
Description:	**Gets all the types which are contained within this type**
Read Only:	Yes
C# Signature:	public ArrayList Types
Remarks:	The *Types* property returns an array list of all of the types which are contained within this type. Each element of the array list is an instance of the TypeContainer class.

C# Example:	public static void ProcessTypes(TypeContainer parent)
	{
	if (parent.Types != null)
	foreach (TypeContainer tc in parent.Types)
	ProcessTypeContainer(tc);
	}

Example Notes:	The example code iterates through each of the type containers contained within the given type and calls the "ProcessTypeContainer" method on each one.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

UnsafeContext

Name:	**UnsafeContext**
Description:	**Determines if this type is defined within an unsafe context**
Read Only:	Yes
C# Signature:	public bool UnsafeContext
Remarks:	The *UnsafeContext* property returns true if the containing type container is unsafe. If the type container is not unsafe the *UnsafeContext* property will return false. A type's unsafe context is determined via the use of the "unsafe" keyword in the type's declaration. For developers who are creating types, the unsafe may be specified by using the following access modifier: Modifiers.UNSAFE Unsafe contexts are required in order to support any operation which involves pointers. When a unsafe context is defined at the type level, all elements defined within that type are said to be unsafe.

C# Example:	public static bool IsUnsafeContext(TypeContainer parent)
	{
	return parent.UnsafeContext;
	}

Example Notes:	The example code uses the *UnsafeContext* property to return true if the given type container contains the "unsafe" modifier. The example method will return false otherwise.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

UserDefinedStaticConstructor

Name:	**UserDefinedStaticConstructor**
Description:	**Returns true if a user defined static constructor is defined on this type**
Read Only:	Yes
C# Signature:	public bool UserDefinedStaticConstructor
Remarks:	The *UserDefinedStaticConstructor* property returns true if a user defined static constructor is defined on this type container, false otherwise. The actual constructor can be retrieved by calling the *DefaultStaticConstructor* method.
C# Example:	public static bool HasUserDefinedStaticConstructor(TypeContainer parent) { return parent.UeserDefinedStaticConstructor; }
Example Notes:	The example code returns true if the given type container has a static constructor defined.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

ValidAttributeTargets

Name:	**ValidAttributeTargets**
Description:	**Returns a string array of valid attribute targets for the given type**
Read Only:	Yes
C# Signature:	public override string[] ValidAttributeTargets
Remarks:	The *ValidAttributeTargets* property returns an array which indicates the type of attributes that can be applied to the containing type. For most types the resultant array will contain a single string with a value of "type".

C# Example:	public static bool IsTypeAttributeTarget(TypeContainer parent)
	{
	string[] targets = parent.ValidAttributeTargets;
	if (targets == null)
	return false;
	foreach (string target in targets)
	if (target == "type")
	return true;
	return false;
	}

Example Notes:	The example code uses the *ValidAttributeTargets* property to get the list of valid attribute targets, and then checks the list to see if one of the attribute targets is a "type". If a type is found the example method will return true. If a"type" target is not found the example method will return false.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Attributes

The following tables will discuss each of the public attributes defined by the TypeContainer type. These attributes can be used to access the structural elements contained within the TypeContainer as well as information about the type container itself.

Basename

Name:	**Basename**
Description:	**A string representation of the base class name**
C# Signature:	public string Basename;
Remarks:	The *Basename* attribute contains a string representation of the base type that the given type container is derived from. In the case where there is no explicit base type, the *Basename* field will contain a null value.
Version Information:	New in Version 0.2.5.1 of the SDK

DocComment

Name:	**DocComment**
Description:	**A string representation of the XML documentation comment**
C# Signature:	public string DocComment;
Remarks:	The *DocComment* attribute contains a string representation of the XML documentation comment which preceded the type container's definition. If no such comment was specified, the *DocComment* attribute will be set to an empty string.
Version Information:	New in Version 0.2.5.1 of the SDK

Kind

Name:	**Kind**
Description:	**Indicates the type of structural element represented by the type container.**
C# Signature:	public readonly Kind Kind;
Remarks:	The *Kind* attribute indicates which type of structural element is represented by the type container. Possible values of the Kind enumeration are as follows: Root Struct Class Interface The only non-intuitive enumeration value is "Root". The "Root" element is used to denote a top level type container which is used as a container for all other type containers being compiled.
Version Information:	New in Version 0.2.5.1 of the SDK

ModFlags

Name:	**ModFlags**
Description:	**Indicates which modifier flags have been applied to the type container**
C# Signature:	public int ModFlags;

Remarks:	The *ModFlags* attribute contains any modifier flags that have been applied to the type container. To determine if a given modifier flag has been applied to the type container, use an and (&) bit mask, against one of the following values:
	
	public const int PROTECTED = 0x0001;
	public const int PUBLIC = 0x0002;
	public const int PRIVATE = 0x0004;
	public const int INTERNAL = 0x0008;
	public const int NEW = 0x0010;
	public const int ABSTRACT = 0x0020;
	public const int SEALED = 0x0040;
	public const int STATIC = 0x0080;
	public const int READONLY = 0x0100;
	public const int VIRTUAL = 0x0200;
	public const int OVERRIDE = 0x0400;
	public const int EXTERN = 0x0800;
	public const int VOLATILE = 0x1000;
	public const int UNSAFE = 0x2000;
	public const int TOP = 0x2000;
	public const int PROPERTY_CUSTOM = 0x4000;
	public const int METHOD_YIELDS = 0x8000;
	public const int METHOD_GENERIC = 0x10000;
	
	The values listed above are defined in the .NET Framework 2.0 Software Development Kit [7].

Version Information:	New in Version 0.2.5.1 of the SDK

NamespaceEntry

Name:	**NamespaceEntry**
Description:	**Contains the namespace entry where the containing type is located**
C# Signature:	public NamespaceEntry NamespaceEntry;
Remarks:	The *NamespaceEntry* attribute contains the namespace entry where the containing type is located. The namespace information can be used for the resolution of other types and to determine other elements that exist in the same namespace.
Version Information:	New in Version 0.2.5.1 of the SDK

Parent

Name:	**Parent**
Description:	**Contains the parent of the type container**
C# Signature:	public DeclSpace Parent;
Remarks:	The *Parent* attribute contains the parent of this type container. If the type has no parent, the *Parent* attribute will be set to null. If the type container is defined within another type, that type will be this type container's parent.
Version Information:	New in Version 0.2.5.1 of the SDK

Pending

Name:	**Pending**
Description:	**Provides a list of the pending methods**
C# Signature:	public PendingImplementation Pending;
Remarks:	The *Pending* attribute contains the pending methods that still need to be implemented before the type container can become concrete. These pending methods can either be from interfaces which are not fully implemented, or from abstract methods without corresponding bodies.
Version Information:	New in Version 0.2.5.1 of the SDK

TypeBuilder

Name:	**TypeBuilder**
Description:	**Points to the actual type definition that is being created**
C# Signature:	public TypeBuilder TypeBuilder;
Remarks:	The *TypeBuilder* attribute contains a reference to the actual type definition that is being created by the type container. The TypeBuilder is part of the System.Reflection.Emit namespace. The type builder will be used when generating the actual MSIL code when re-assembling the SUT. Developers can use the *TypeBuilder* field to work directly with the code generation process or add program elements via the types defined within the System.Reflection.Emit namespace. For more information please see the .NET Framework 2.0 Software Development Kit (SDK) [7].
Version Information:	New in Version 0.2.5.1 of the SDK

Field

A static extension may wish to operate and examine a field defined within the SUT. These fields will be bound to an identifier contained within the contract language via the Contract Editor's Binding Tool [2]. These types are represented by the Field class [6]. The Field class is defined within the CSCompiler.dll file, which is included as part of the Contract Evaluation Engine Extension SDK. The Field class is located in the "DaveArnold.Contracts.CSCompiler" namespace.

The Field class is contained within the CSCompiler.dll file, rather than the ExtensionAPI.dll file, because the Field class represents a field defined within the SUT, and this field will be emitted by the C# compiler for profiling purposes. As the emission takes place following the execution of static checks, user defined static extensions may modify any aspect of the field.

Note: Implementers of static extensions which modify the SUT should do so with caution. If a modification results in an error while recompiling the executable, the compile-time error will be presented to the user. Such an error will prevent the SUT from profiled and the contract evaluation process will fail.

The following tables list all of the relevant, from the static extension point of view, methods and properties defined within the Field type. The following tables intentionally omit a few methods defined in the Field type. The methods that are omitted should not be used by a static extension. These methods perform activities that should only be executed by the C# compiler once all of the static checks have been completed. Some of the method, properties, and fields discussed below are based on the AST data types found in the Mono C# compiler [6].

CheckObsoleteness

<table>
<tr><td>Name:</td><td>CheckObsoleteness</td></tr>
<tr><td>Description:</td><td>Checks to see if the containing field is marked as obsolete</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Location</td><td>The source code location where the check is based</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The CheckObsoleteness method is used to check if the containing field is marked as obsolete. Marking a field as obsolete consists of using the “Obsolete” attribute before the field definition.

Obsolete program elements will issue a compile time warning when they are referenced in code. The idea is to mark a given program element as obsolete to discourage its usage and to indicate to the user that such element should no longer be used.

The CheckObsoleteness method uses the C# compiler error reporting mechanism to indicate if a field is marked as obsolete or not. The example, will present a method of creating a Boolean wrapper method.</td></tr>
<tr><td>C# Example:</td><td>
<pre>
public static bool IsObsolete(Field field)
{
    bool result = false;
    int warnCount = Report.Warnings;
    System.IO.TextWriter writer = Report.Stderr;
    Report.Stderr = null;
    field.CheckObsoleteness(field.Location);
    if (Report.Warnings != warnCount)
    {
        result = true;
        Report.Warnings = warnCount;
    }
    Report.Stderr = writer;
    return result;
}
</pre>
</td></tr>
<tr><td>Example Notes:</td><td>The example method begins by saving the current number of warnings, and the error reporting stream. The method then, sets the error reporting stream to null. The purpose of this is to suppress the reporting of the warning to the user. After calling the CheckObsoleteness method, if the number of warnings has changed, then an “Obsolete” attribute has been found. The example method finishes, by resetting the error reporting object, and returning the result.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public virtual void CheckObsoleteness(Location loc);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

GetDocCommentName

Name:	**GetDocCommentName**
Description:	**Returns a string that represents the signature for the containing field**
Parameters:	1
Parameter Values:	Ordinal: 1 — Type: DeclSpace — Description: The containing declaration space
Remarks:	The *GetDocCommentName* method returns a string that represents the signature for the given field. This string should be used for generating XML documentation or when writing information to the Contract Evaluation Report. The "DeclSpace" parameter specifies the containing declaration space for the field.

C# Example:	public static string GetDocName(Field field)
	{
	return field.GetDocCommentName(field.Parent);
	}

Example Notes:	The example method gets the document comment name for the given field. The field's parent contains the containing declaration space.
Thread Safe:	No
C# Method Signature:	public virtual string GetDocCommentName(DeclSpace ds);
Version Information:	New in Version 0.2.5.1 of the SDK

GetObsoleteAttribute

Name:	**GetObsoleteAttribute**
Description:	**Gets an instance of the obsolete attribute for this field**
Parameters:	o
Parameter Values:	N/A
Remarks:	The *GetObsoleteAttribute* method returns an instance of the obsolete attribute if one is applied to the containing field. If no such attribute is applied the *GetObsoleteAttribute* method returns null. Obsolete program elements will issue a compile time warning when they are referenced in code. The idea is to mark a given program element as obsolete to discourage its usage and to indicate to the user that such element should no longer be used. The obsolete attribute, is defined by the "ObsoleteAttribute" class. This class is part of the .NET Framework SDK. For more information please see the SDK documentation [7].

C# Example:	public static bool IsObsolete(Field field)
	{
	ObsoleteAttribute attr = field.GetObsoleteAttribute();
	return (attr != null);
	}

Example Notes:	The example method uses the *GetObsoleteAttribute* method to get the "ObsoleteAttribute" instance if availiable. The example method will return true if the given field is marked with the Obsolete attribute.
Thread Safe:	No
C# Method Signature:	public virtual ObsoleteAttribute GetObsoleteAttribute();
Version Information:	New in Version 0.2.5.1 of the SDK

GetSignatureForError

Name:	**GetSignatureForError**
Description:	**Gets a string representation of the field's signature**
Parameters:	0
Parameter Values:	N/A
Remarks:	The *GetSignatureForError* method returns a string representation of the field's signature. This signature can be used for generating errors which reference the field. The method can also be used for generating messages to return to the contract evaluation engine regarding the field. This method is included for developer assistance. The *GetSignatureForError* method does not have any functional purpose, and does not modify the field in any way.
C# Example:	`public static bool VerifyClsCompliance(Field field)` `{` `  if (!field.IsClsComplianceRequired(field.Parent))` `  {` `  if (field.HasClsCompliantAttribute &&` `  RootContext.WarningLevel >= 2)` `  {` `  if (!field.IsExposedFromAssembly(field.Parent))` `  Report.Warning(3019, 2, field.Location,` `  "CLS compliance checking will" +` `  "not be performed on `{0}'" +` ` "because it is not visible from" +` ` "outside this assembly",` ` field.GetSignatureForError());` ` if (!CodeGen.Assembly.IsClsCompliant)` ` Report.Warning(3021, 2, field.Location,` ` "`{0}' does not need a" +` `  "CLSCompliant attribute" +` `  "because the assembly is not" +` `  "marked as CLS-compliant",` `  field.GetSignatureForError());` `  }` `  return false;` `  }` `  if (!CodeGen.Assembly.IsClsCompliant)` `  {` `  if (field.HasClsCompliantAttribute)` `  {` `  Report.Error(3014, field.Location,` `  "`{0}' cannot be marked as" +` ` "CLS-compliant because the assembly" +` ` "is not marked as CLS-compliant",` ` field.GetSignatureForError());` ` }

	return false;
	}
	if (field.MemberName.Name[0] == '_')
	{
	Report.Error(3008, field.Location,
	"Identifier `{0}' is not CLS-compliant",
	field.GetSignatureForError());
	}
	return true;
	}

Example Notes:	The example method above checks to see if the given field is CLS Compliant. If an error is encountered the *GetSignatureForError* method is used to get a string representation of the field.
Thread Safe:	Yes
C# Method Signature:	public override string GetSignatureForError();
Version Information:	New in Version 0.2.5.1 of the SDK

IsClsComplianceRequired

<table>
<tr><td>Name:</td><td>IsClsComplianceRequired</td></tr>
<tr><td>Description:</td><td>Checks to see if this field must be CLS Compliant</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>DeclSpace</td><td>The containing type</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The IsClsComplianceRequired method performs checks to see if the containing field is required to be CLS Compliant or not.

The “DeclSpace” parameter is used to specify the containing type which contains the field. The IsClsComplianceRequired will return true if the field must be CLS Compliant, false otherwise.

If you are writing .NET classes, which will be used by other .NET classes irrespective of the language they are implemented, then your code should conform to the CLS (Common Language Specification). This means that your class should only expose features that are common across all .NET languages. The following are the basic rules that should be followed when writing CLS complaint C# code.

1. Unsigned types should not be part of the public interface of the class. What this means is public fields should not have unsigned types like uint or ulong, public methods should not return unsigned types, parameters passed to public function should not have unsigned types. However unsigned types can be part of private members.
2. Unsafe types like pointers should not be used with public members. However they can be used with private members.
3. Class names and member names should not differ only based on their case. For example you cannot have two methods named MyMethod and MYMETHOD.
4. Only properties and methods may be overloaded. Operators should not be overloaded.

For more information on CLS Compliance please see the .NET Framework 2.0 SDK documentation [7].</td></tr>
<tr><td>C# Example:</td><td>public static bool MustBeClsCompliant(Field field)
{
    return field.IsClsComplianceRequired(field.Parent);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method uses the IsClsComplianceRequired method to determine if the given field is required to be CLS Compliant or not.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public override bool IsClsComplianceRequired(DeclSpace container);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

IsExposedFromAssembly

Name:	**IsExposedFromAssembly**
Description:	**Checks to see if this field is exposed from the assembly**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>DeclSpace</td><td>The containing type if any</td></tr></table>
Remarks:	The *IsExposedFromAssembly* method performs checks to see if field is exposed from the assembly or not. Being exposed from an assembly simply means that program elements that reside outside of the field's assembly are able to view the field. That is, the field's accessibility modifiers allow it to be seen outside of the assembly. The "DeclSpace" parameter is used to specify the containing type which contains the field. The *IsExposedFromAssembly* will return true if the field is visible from outside the assembly, false otherwise.

C# Example:	public static bool IsFieldExposed(Field field)
	{
	return field.IsExposedFromAssembly(field.Parent);
	}

Example Notes:	The example method uses the *IsTypeContainerExposed* method to determine if the given field is visible from outside the assembly or not.
Thread Safe:	No
C# Method Signature:	public bool IsExposedFromAssembly(DeclSpace ds);
Version Information:	New in Version 0.2.5.1 of the SDK

IsObsolete

Name:	**IsObsolete**
Description:	**Checks to see if this field is marked as obsolete**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>TypeContainer</td><td>The containing type</td></tr></table>
Remarks:	The *IsObsolete* method performs checks to see if field is marked as obsolete or not. A field is marked obsolete if the obsolete attribute is applied to the field. Obsolete program elements will issue a compile time warning when they are referenced in code. The idea is to mark a given program element as obsolete to discourage its usage and to indicate to the user that such element should no longer be used. The "TypeContainer" parameter is used to specify the containing type which contains the field. The *IsObsolete* method will return true if the field is marked as obsolete, false otherwise.

C# Example:	`public static bool IsFieldObsolete(Field field)`
	`{`
	`  return field.IsObsolete(field.Parent);`
	`}`

Example Notes:	The example method uses the *IsObsolete* method to determine if the given field is marked as obsolete or not.
Thread Safe:	No
C# Method Signature:	`public bool IsObsolete(TypeContainer tc);`
Version Information:	New in Version 0.2.5.1 of the SDK

SetAssigned

Name:	**SetAssigned**
Description:	**Sets this field as assigned**
Parameters:	o
Parameter Values:	N/A
Remarks:	The *SetAssigned* method sets a flag within the field to mark the field as assigned. The assigned field is used to ensure that a field is assigned before it is used. When adding fields via a static extension, developers can use the *SetAssigned* method to mark newly created fields as assigned. Fields which are not assigned will receive a corresponding warning message from the C# compiler, when the SUT is rebuilt following the execution of static checks. Use the *SetAssigned* method to prevent the warning message.

C# Example:	public static void MarkFieldAsAssigned(Field field)
	{
	field.SetAssigned();
	}

Example Notes:	The example method uses the *SetAssigned* method to mark the given field as assigned. Assigned fields will not generate unassigned field warning messages.
Thread Safe:	Yes
C# Method Signature:	public void SetAssigned();
Version Information:	New in Version 0.2.5.1 of the SDK

SetMemberIsUsed

Name:	**SetMemberIsUsed**
Description:	**Marks the field as being used**
Parameters:	o
Parameter Values:	N/A
Remarks:	The *SetMemberIsUsed* method sets an internal flag in the field to mark it as used within the current code base. Normally, a program element's usage is calculated automatically by the C# compiler. However, sometimes when adding code via a static extension, the compiler may indicate that a given program element is not used. This method will allow you to remove that warning from the compiler output.

C# Example:	public static void MarkAsUsed(Field field)
	{
	field.SetMemberIsUsed();
	}

Example Notes:	The example method uses the *SetMemberIsUsed* method to mark the given field as used.
Thread Safe:	Yes
C# Method Signature:	public void SetMemberIsUsed();
Version Information:	New in Version 0.2.5.1 of the SDK

UnsafeOK

<table>
<tr><td>Name:</td><td>UnsafeOK</td></tr>
<tr><td>Description:</td><td>Checks to see if this field can use pointers</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>DeclSpace</td><td>The containing type</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The UnsafeOK method performs checks to see if it is okay to use an unsafe pointer within the field. That is the field can be a pointer to something.

The “DeclSpace” parameter is used to specify the containing type which contains the field.

The UnsafeOK method will return true if the field can use unsafe pointers, false otherwise.</td></tr>
<tr><td>C# Example:</td><td>public static bool CanUsePointers(Field field)
{
    return field.UnsafeOK(field.Parent);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method uses the UnsafeOK method to determine if the given field can use unsafe pointers or not.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public bool UnsafeOK(DeclSpace parent);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

Properties

The following tables will discuss each of the properties defined by the Field type. These properties can be used to access the structural elements contained within the Field.

AttributeTargets

Name:	**AttributeTargets**
Description:	**Gets the type of attributes which can be applied to the field**
Read Only:	Yes
C# Signature:	public AttributeTargets AttributeTargets
Remarks:	The *AttributeTargets* property returns a System.AttributeTargets enumeration value indicating the type of attributes which can be applied to this field. The only possible return value is as follows: AttributeTargets.Field;

C# Example:	public static AttributeTargets GetAttributeTargets(Field field)
	{
	return field.AttributeTargets;
	}

Example Notes:	Uses the property to get the type of attributes which can be applied to the given field.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

DocCommentHeader

Name:	**DocCommentHeader**
Description:	**Gets the xml document header string for xml comment output**
Read Only:	Yes
C# Signature:	public override string DocCommentHeader
Remarks:	The *DocCommentHeader* property returns the xml document header string for xml comment output. This property is only used for xml comment generation. It is included in the Contract Evaluation Engine Extension SDK so that if xml code comments are being generated as part of static evaluation the necessary API is available.

C# Example:	public static string GetDocCommentName(Field field)
	{
	return String.Concat(field.DocCommentHeader,field.Name);
	}

Example Notes:	The example code returns the correct xml comment name for the given field.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Initializer

Name:	**Initializer**
Description:	**Sets the expression which will be used to initialize the field**
Read Only:	Write only
C# Signature:	public Expression Initializer
Remarks:	The *Initializer* property is used to set an expression that will be used to initialize the field. This property is write only, and once a field has been assigned an initializer, it should not be removed. That is, never set the *Initializer* property to null.

C# Example:	public static void SetToZero(Field field)
	{
	field.Initializer = new IntLiteral(0, field.Location);
	}

Example Notes:	The example code sets the initializer for the given field to a value of zero. The example assumes that the given field will be a type that can be assigned an integer value.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

InUnsafe

Name:	**InUnsafe**
Description:	**Determines if this field is marked as unsafe**
Read Only:	Yes
C# Signature:	public bool InUnsafe
Remarks:	The *InUnsafe* property indicates if the given field is marked as unsafe. Unsafe fields are able to use pointers. A field is marked as unsafe by using the "unsafe" keyword as a modifier when the field is defined.

C# Example:	public static bool IsFieldUnsafe(Field field)
	{
	return field.InUnsafe;
	}

Example Notes:	The example code uses the *InUnsafe* property to determine if the given field is marked as unsafe or not.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

IsUsed

Name:	**IsUsed**
Description:	**Determines if this field is used or not**
Read Only:	Yes
C# Signature:	public virtual bool IsUsed
Remarks:	The *IsUsed* property indicates if the given field is used or referenced by another program element. The property will return true if the field is used by another program element, false otherwise. Developers can mark a field as used via the "SetMemberIsUsed" method. The method has been previously discussed.

C# Example:	public static bool IsFieldUsed(Field field)
	{
	return field.IsUsed;
	}

Example Notes:	The example code uses the *IsUsed* property to determine if the given field is used or referenced by another program element.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Location

Name:	**Location**
Description:	**Gets the source code location where the field was defined**
Read Only:	Yes
C# Signature:	public Location Location
Remarks:	The *Location* property represents the source location where the field was defined. That is, the source code file name and line number. The line number specified is where the first line of the field's definition is located. The "Location" class contains several members to access the source code file, line number, and column where the field's definition is located.

C# Example:	public static int GetFieldDefinitionLineNumber(Field field)
	{
	return field.Location.Row;
	}

Example Notes:	The example code uses the *Location* property to access the source code location where the field was defined. Once the location information is retrieved, the example code returns the line number (row) where the definition is located.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

MemberName

Name:	**MemberName**
Description:	**Gets the member name object that represents the field**
Read Only:	Yes
C# Signature:	public MemberName MemberName
Remarks:	The *MemberName* property returns a member name object which represents the name of the field. The "MemberName" type contains the name, namespace, and source code location where the field has been defined. Each of these elements can be accessed via dedicated properties defined on the "MemberName" type. If you are only interested in the name of the field, use the "Name" property instead.

C# Example:	public static string GetFullName(Field field)
	{
	return field.MemberName.GetName();
	}

Example Notes:	The example code uses the *MemberName* property to get the member name object, and then calls the "GetName" method defined on the "MemberName" type. The "GetName" method returns a string representation of the fully resolved name of the given field.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

MemberType

Name:	**MemberType**
Description:	**Gets the field's type**
Read Only:	Yes
C# Signature:	public Type MemberType
Remarks:	The *MemberType* property returns a System.Type object that represents the field's type. If the field's type has not yet been resolved, the property will resolve the field's type. If such a resolution cannot be performed, the property will return null. For more information on the System.Type class, please see the .NET Framework 2.0 documentation [7].

C# Example:	public static Type GetFieldType(Field field)
	{
	return field.MemberType;
	}

Example Notes:	The example code uses the *MemberType* property to get the System.Type object that represents the field's type.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Name

Name:	**Name**
Description:	**Gets the name of this field**
Read Only:	Yes
C# Signature:	public string Name
Remarks:	The *Name* property returns a string representation of the field's name. This name will include namespace information as well as the field's actual name. The *Name* property yields the same result as calling "MemberName.GetName(false)" however, the name is cached so the *Name* property should be used instead.

C# Example:	public static string GetFieldName(Field field)
	{
	return field.Name;
	}

Example Notes:	The example code uses the *Name* property to get the full name of the given field.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

OptAttributes

Name:	**OptAttributes**
Description:	**Contains a list of all the attributes which have been applied to this field**
Read Only:	No
C# Signature:	public Attributes OptAttributes
Remarks:	The *OptAttributes* property contains a list of all the attributes which have been applied to this field. The *OptAttributes* property also allows for new attributes to be applied programmatically.

C# Example:	public static void AddAttribute(Field field, Attribute newAttribute)
	{
	if (field.OptAttributes == null)
	field.OptAttributes = new Attributes(newAttribute);
	else
	field.OptAttributes.Attrs.Add(newAttribute);
	}

Example Notes:	The example code adds a new attribute to the given field. An example of such usage would be the addition of the previously discussed "Obsolete" attribute to the given field. The example code supports two cases. The first case, is where the new attribute is the first attribute to be added to the field. In that case, a new "Attributes" container is created and the new attributed is added via the construction. The second case, is where the new attribute is being added to the existing list of attributes. This is accomplished via the "Add" method, defined on the actual attribute list. The actual attribute list is exposed via the "Attrs" property.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Parent

Name:	**Parent**
Description:	**Contains the type container where the field has been defined**
Read Only:	Yes
C# Signature:	public new TypeContainer Parent
Remarks:	The *Parent* property contains the type container where the field has been defined. This value cannot be null, as C# does not permit global fields.

C# Example:	public static TypeContainer GetFieldContainer(Field field)
	{
	  return field.Parent;
	}

Example Notes:	The example code gets the TypeContainer object that contains the given field. That is, the type container where the field is defined is returned.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

ShortName

Name:	**ShortName**
Description:	**Contains a string representation of the field's name without the namespace or explicit interface definition name**
Read Only:	No
C# Signature:	public string ShortName
Remarks:	The *ShortName* property contains a string representation of the field's name. This name does not contain any namespace or containing type information. In addition, the name does not contain the explicit interface definition name, if one exists. The *ShortName* property can be written to. This will effectively rename the field. It will not modify any code which references the field.

C# Example:	public static void RenameField(Field field, string newName)
	{
	field.ShortName = newName;
	}

Example Notes:	The example code uses the *ShortName* property to rename the given field. This code will not modify any body code that may make reference to the field.
Thread Safe:	Read – Yes, Write – No
Version Information:	New in Version 0.2.5.1 of the SDK

ValidAttributeTargets

Name:	**ValidAttributeTargets**
Description:	**Returns a string array of valid attribute targets for the given field**
Read Only:	Yes
C# Signature:	public override string[] ValidAttributeTargets
Remarks:	The *ValidAttributeTargets* property returns an array which indicates the type of attributes that can be applied to the field. For fields the resultant array will contain a single string with a value of "field".

C# Example:	public static bool IsFieldAttributeTarget(Field field)
	{
	string[] targets = field.ValidAttributeTargets;
	if (targets == null)
	return false;
	foreach (string target in targets)
	if (target == "field")
	return true;
	return false;
	}

Example Notes:	The example code uses the *ValidAttributeTargets* property to get the list of valid attribute targets, and then checks the list to see if one of the attribute targets is a "field". If a field is found the example method will return true. If a"field" target is not found the example method will return false.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Attributes

The following tables will discuss each of the public attributes defined by the Field type. These attributes can be used to access the elements contained within the field as well as information about the field itself.

conflict_symbol

Name:	**conflict_symbol**
Description:	**The symbol with the same name in the base class or structure**
C# Signature:	public MemberInfo conflict_symbol;
Remarks:	The *conflict_symbol* attribute contains the MemberInfo object of the symbol with the same name as in the base class or structure. That is, the element that this field will override. If the *conflict_symbol* attribute is set to null, then no such override will occur.
Version Information:	New in Version 0.2.5.1 of the SDK

DocComment

Name:	**DocComment**
Description:	**A string representation of the XML documentation comment**
C# Signature:	public string DocComment;
Remarks:	The *DocComment* attribute contains a string representation of the XML documentation comment which preceded the field's definition. If no such comment was specified, the *DocComment* attribute will be set to an empty string.
Version Information:	New in Version 0.2.5.1 of the SDK

FieldBuilder

Name:	**FieldBuilder**
Description:	**Points to the actual field definition that is being created**
C# Signature:	public FieldBuilder FieldBuilder;
Remarks:	The *FieldBuilder* attribute contains a reference to the actual field definition that is being created by the field. The FieldBuilder is part of the System.Reflection.Emit namespace. The field builder will be used when generating the actual MSIL code when re-assembling the SUT. Developers can use the *FieldBuilder* field to work directly with the code generation process via the types defined within the System.Reflection.Emit namespace. For more information please see the .NET Framework 2.0 Software Development Kit (SDK) [7].
Version Information:	New in Version 0.2.5.1 of the SDK

InterfaceType

Name:	**InterfaceType**
Description:	**The interface type that we are explicitly implementing**
C# Signature:	public Type InterfaceType;
Remarks:	The *InterfaceType* attribute contains the type of interface that we are explicitly implementing. If no such interface exists, then the *InterfaceType* attribute will be set to null. For more information on the System.Type class please see the .NET Framework 2.0 Software Development Kit (SDK) [7]. **Note**: In the case of Fields the *InterfaceType* attribute will always be set to null.
Version Information:	New in Version 0.2.5.1 of the SDK

IsExplicitImpl

Name:	**IsExplicitImpl**
Description:	**True if we are explicitly implementing an interface**
C# Signature:	public bool IsExplicitImpl;
Remarks:	The *IsExplicitImpl* attribute indicates if the original SUT code is explicitly implementing an interface. If there is no explicit interface implementation then the *IsExplicitImpl* attribute will be set to false. **Note**: In the case of Fields the *IsExplicitImpl* attribute will always be set to false.
Version Information:	New in Version 0.2.5.1 of the SDK

IsInterface

Name:	**IsInterface**
Description:	**True if we are an interface member**
C# Signature:	public bool IsInterface;
Remarks:	The *IsInterface* attribute indicates if this field is an interface member. If the *IsInterface* attribute is set to false, it indicates that we are not part of an interface member. **Note**: In the case of Fields the *IsInterface* attribute will always be set to false.
Version Information:	New in Version 0.2.5.1 of the SDK

ModFlags

Name:	**ModFlags**
Description:	**Indicates which modifier flags have been applied to the field**
C# Signature:	public int ModFlags;

Remarks:	The *ModFlags* attribute contains any modifier flags that have been applied to the field. To determine if a given modifier flag has been applied to the field, use an and (&) bit mask, against one of the following values:
	
	public const int PROTECTED = 0x0001;
	public const int PUBLIC = 0x0002;
	public const int PRIVATE = 0x0004;
	public const int INTERNAL = 0x0008;
	public const int NEW = 0x0010;
	public const int ABSTRACT = 0x0020;
	public const int SEALED = 0x0040;
	public const int STATIC = 0x0080;
	public const int READONLY = 0x0100;
	public const int VIRTUAL = 0x0200;
	public const int OVERRIDE = 0x0400;
	public const int EXTERN = 0x0800;
	public const int VOLATILE = 0x1000;
	public const int UNSAFE = 0x2000;
	public const int TOP = 0x2000;
	public const int PROPERTY_CUSTOM = 0x4000;
	public const int METHOD_YIELDS = 0x8000;
	public const int METHOD_GENERIC = 0x10000;
	
	The values listed above are defined in the .NET Framework 2.0 Software Development Kit [7].

Version Information:	New in Version 0.2.5.1 of the SDK

Type

Name:	**Type**
Description:	**The expression that represents the field's type**
C# Signature:	public Expression Type;
Remarks:	The *Type* attribute contains the expression that was used to define the field's type. To get a System.Type representation of the Field's type, use the previously discussed "MemberType" property.
Version Information:	New in Version 0.2.5.1 of the SDK

Integer

The Integer type represents an integral value. Integer values are signed, and are 32-bits in length. They have a range of -2,147,483,648 to 2,147,483,647. To specify a static entry point that takes a parameter of the Integer type, use the .NET System.Int32 structure. This structure is aliased by the *int* keyword in most .NET languages. Figure 12, illustrates a static extension that takes one parameter of the Integer type.

```
[StaticEntryPoint]
public StaticEvaluationResult DoEvaluation(TypeContainer type, int iValue)
{
        // To Do: Put code to implement the static extension here
}
```

Figure 12 – Static Entry Point Signature with an Integer Parameter

Method

A static extension may wish to operate and examine a method defined within the SUT. These methods will be bound to an identifier contained within the contract language via the Contract Editor's Binding Tool [2]. These methods are represented by the Method class [6]. The Method class is defined in the CSCompiler.dll file, which is included as part of the Contract Evaluation Engine Extension SDK. The Method class is located in the "DaveArnold.Contracts.CSCompiler" namespace.

The Method class is contained within the CSCompiler.dll file, rather than the ExtensionAPI.dll file, because the Method class represents a method defined within the SUT, and this method will be emitted by the C# compiler for profiling purposes. As the emission takes place following the execution of static checks, user defined static extensions may modify any aspect of the method. Such modification may include the addition of code to check pre- and post-conditions, adding specialized code to provide profiler instruction, and reporting tools for runtime analysis or debugging.

Note: Implementers of static extensions which modify the SUT should do so with caution. If a modification results in an error while recompiling the executable, the compile-time error will be presented to the user. Such an error will prevent the SUT from profiled and the contract evaluation process will fail.

The following tables, list all of the relevant methods, properties, and attributes defined within the Method type. The following tables intentionally omit a few methods defined in the Method type. The methods that are omitted should not be used by a static extension. These methods perform activities that should only be executed by the C# compiler once all of the static checks have been completed. An example of such a method is the Emit method which will emit the actual MSIL code for the method and will be called by the C# compiler during the code generation phase. Some of the method, properties, and fields discussed below are based on the AST data types found in the Mono C# compiler [6].

CheckAbstractAndExtern

<table>
<tr><td>Name:</td><td>CheckAbstractAndExtern</td></tr>
<tr><td>Description:</td><td>Ensures that all abstract and external methods are handled correctly</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Boolean</td><td>True if the method contains a top level code block, false otherwise.</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The CheckAbstractAndExtern method examines the method's modifier flags and determines if the method should contain a method body or not. This result is compared with the given parameter, and if they do not match, the appropriate error is generated. That is, if an abstract or external method defines a method body, an error will be generated. In addition, if a method that is not marked as abstract or external does not define a method body a corresponding error will be generated.

Developers can use the CheckAbstractAndExtern method to verify that any methods or method bodies added via a static extension are marked correctly with respect to abstract and external modifiers.

For more information on the "abstract" and "external" keyword, please see the .NET Framework 2.0 SDK documentation [7].</td></tr>
<tr><td>C# Example:</td><td>
<pre>
public static bool IsAbstractAndExternCorrect(Method method)
{
    bool result = false;
    int errorCount = Report.Errors;
    System.IO.TextWriter writer = Report.Stderr;
    Report.Stderr = null;
    method.CheckAbstractAndExtern((method.Block != null));
    if (Report.Errors != errorCount)
    {
        result = true;
        Report.Errors = errorCount;
    }
    Report.Stderr = writer;
    return result;
}
</pre>
</td></tr>
<tr><td>Example Notes:</td><td>The example method begins by saving the current number of errors, and the error reporting stream. The method then, sets the error reporting stream to null. The purpose of this is to suppress the reporting of any errors to the user. After calling the CheckAbstractAndExtern method, if the number of errors has changed, then there is an error with the "abstract" and/or "extern" modifier(s). The example method finishes, by resetting the error reporting object, and returning the result.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public bool CheckAbstractAndExtern(bool has_block);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

CheckObsoleteness

<table>
<tr><td>Name:</td><td>CheckObsoleteness</td></tr>
<tr><td>Description:</td><td>Checks to see if the method is marked as obsolete</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Location</td><td>The source code location where the check is based</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The CheckObsoleteness method is used to check if this method is marked as obsolete. Marking a method as obsolete consists of using the “Obsolete” attribute before the method definition.

Obsolete program elements will issue a compile time warning when they are referenced in code. The idea is to mark a given program element as obsolete to discourage its usage and to indicate to the user that such element should no longer be used.

The CheckObsoleteness method will also examine base methods. That is, if the method is not marked as obsolete, but one of its bases is marked as obsolete, the method will report an obsolete warning message.

The CheckObsoleteness method uses the C# compiler error reporting mechanism to indicate if a method is marked as obsolete or not. The example, will present a method of creating a Boolean wrapper method.</td></tr>
<tr><td>C# Example:</td><td>
<pre>
public static bool IsObsolete(Method method)
{
    bool result = false;
    int warnCount = Report.Warnings;
    System.IO.TextWriter writer = Report.Stderr;
    Report.Stderr = null;
    method.CheckObsoleteness(method.Location);
    if (Report.Warnings != warnCount)
    {
        result = true;
        Report.Warnings = warnCount;
    }
    Report.Stderr = writer;
    return result;
}
</pre>
</td></tr>
</table>

Example Notes:	The example method begins by saving the current number of warnings, and the error reporting stream. The method then, sets the error reporting stream to null. The purpose of this is to supress the reporting of the warning to the user. After calling the *CheckObsoleteness* method, if the number of warnings has changed, then an "Obsolete" attribute has been found. The example method finishes, by resetting the error reporting object, and returning the result.
Thread Safe:	No
C# Method Signature:	public virtual void CheckObsoleteness(Location loc);
Version Information:	New in Version 0.2.5.1 of the SDK

Error1599

<table>
<tr><td>Name:</td><td>Error1599</td></tr>
<tr><td>Description:</td><td>Generates C# compiler error CS1599</td></tr>
<tr><td>Parameters:</td><td>2</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>Location</td><td>The source code location where the error occurs</td></tr>
<tr><td>2</td><td>Type</td><td>The System.Type instance that caused the error</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The Error1599 method will generate C# compiler error CS1599. Error CS1599 is generated when a method or delegate returns a type which is not allowed. According to the .NET Framework 2.0 SDK there are some types defined in the base class library that cannot be returned. An example of these types is the TypedReference, or the ArgIterator types.

This method will generate a compiler error, and should only be used if such an error is desired. The method does not check for or detect the error, it simply reports it. The first parameter, the location, indicates where in the source code the error occurs. This usually is the method's location value. The method's location value can be acquired via the "Location" property. The second parameter, the type, is the type which the method is trying to return.

For more information on which types cannot be returned, please see the .NET Framework 2.0 SDK documentation [7].</td></tr>
<tr><td>C# Example:</td><td>public static void GenerateError1599(Method method)
{
    Method.Error1599(method.Location, method.ReturnType);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method uses the Error1599 static method to generate the C# compiler error CS1599 for the given method. The method's location and return type are used to generate the error.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public static void Error1599(Location loc, Type t);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

GetDocCommentName

<table>
<tr><td>Name:</td><td>GetDocCommentName</td></tr>
<tr><td>Description:</td><td>Returns a string that represents the signature for the method</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>DeclSpace</td><td>The containing declaration space</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The GetDocCommentName method returns a string that reprsents the signature for the given method. This string should be used for generating XML documentation or when writing information to the Contract Evaluation Report.

The “DeclSpace” parameter specifies the containing declaration space, if any. If there is no containing declaration space, then a value of “null” should be used. As C# does not support global methods, all methods defined in C# will contain a valid declaration space.</td></tr>
<tr><td>C# Example:</td><td>public static string GetDocName(Method method)
{
    return method.GetDocCommentName(method.Parent);
}</td></tr>
<tr><td>Example Notes:</td><td>The example code gets the document comment name for the given method.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public virtual string GetDocCommentName(DeclSpace ds);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

GetObsoleteAttribute

Name:	**GetObsoleteAttribute**
Description:	**Gets an instance of the obsolete attribute for this method**
Parameters:	o
Parameter Values:	N/A
Remarks:	The *GetObsoleteAttribute* method returns an instance of the obsolete attribute if one is applied to the containing method. If no such attribute is applied the *GetObsoleteAttribute* method returns null. Obsolete program elements will issue a compile time warning when they are referenced in code. The idea is to mark a given program element as obsolete to discourage its usage and to indicate to the user that such element should no longer be used. The obsolete attribute, is defined by the "ObsoleteAttribute" class. This class is part of the .NET Framework SDK. For more information please see the SDK documentation [7].

C# Example:	public static bool IsObsolete(Method method)
	{
	ObsoleteAttribute attr = method.GetObsoleteAttribute();
	return (attr != null);
	}

Example Notes:	The example method uses the *GetObsoleteAttribute* method to get the "ObsoleteAttribute" instance if availiable. The example method will return true if the given method is marked with the Obsolete attribute.
Thread Safe:	No
C# Method Signature:	public virtual ObsoleteAttribute GetObsoleteAttribute();
Version Information:	New in Version 0.2.5.1 of the SDK

GetSignatureForError

Name:	**GetSignatureForError**
Description:	**Gets a string representation of the method's signature**
Parameters:	0
Parameter Values:	N/A
Remarks:	The *GetSignatureForError* method returns a string representation of the method's signature. This signature can be used for generating errors which reference the method. The *GetSignatureForError* method can also be used for generating messages to return to the contract evaluation engine regarding the method. This method is included for developer assistance. The *GetSignatureForError* method does not have any functional purpose, and does not modify the method in any way.

C# Example:	public static bool VerifyClsCompliance(Method method) { if (!method.VerifyClsCompliance(method.Parent)) { if ((method.ModFlags & Modifiers.ABSTRACT) != 0 && method.IsExposedFromAssembly(method.Parent) && method.Parent.IsClsComplianceRequired(method.Parent)) { Report.Error(3011, method.Location, "`{0}': only CLS-compliant members can be abstract", method.GetSignatureForError()); } return false; } if (method.Parameters.HasArglist) { Report.Error(3000, method.Location, "Methods with variable arguments are not CLS- compliant"); } if (!AttributeTester.IsClsCompliant(method.MemberType)) { if (method is PropertyBase) Report.Error(3003, method.Location, "Type of `{0}' is not CLS-compliant", method.GetSignatureForError()); else Report.Error(3002, method.Location, "Return type of `{0}' is not CLS-compliant", method,GetSignatureForError()); } method.Parameters.VerifyClsCompliance(); return true; }
Example Notes:	The example method above checks to see if the given method is CLS Compliant. If an error is encountered the *GetSignatureForError* method is used to get a string representation of the method.
Thread Safe:	No
C# Method Signature:	public override string GetSignatureForError();
Version Information:	New in Version 0.2.5.1 of the SDK

IsClsComplianceRequired

Name:	**IsClsComplianceRequired**
Description:	**Checks to see if this method must be CLS Compliant**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>DeclSpace</td><td>The containing type</td></tr></table>
Remarks:	The *IsClsComplianceRequired* method performs checks to see if the method is required to be CLS Compliant or not. The "DeclSpace" parameter is used to specify the containing type which contains the method. The *IsClsComplianceRequired* will return true if the method must be CLS Compliant, false otherwise. If you are writing .NET classes, which will be used by other .NET classes irrespective of the language they are implemented, then your code should conform to the CLS (Common Language Specification). This means that your class should only expose features that are common across all .NET languages. The following are the basic rules that should be followed when writing CLS complaint C# code. 1. Unsigned types should not be part of the public interface of the class. What this means is public fields should not have unsigned types like uint or ulong, public methods should not return unsigned types, parameters passed to public function should not have unsigned types. However unsigned types can be part of private members. 2. Unsafe types like pointers should not be used with public members. However they can be used with private members. 3. Class names and member names should not differ only based on their case. For example you cannot have two methods named MyMethod and MYMETHOD. 4. Only properties and methods may be overloaded. Operators should not be overloaded. For more information on CLS Compliance please see the .NET Framework 2.0 SDK documentation [7].

C# Example:	`public static bool MustBeClsCompliant(Method method)`
	`{`
	`  return method.IsClsComplianceRequired(method.Parent);`
	`}`

Example Notes:	The example method uses the *IsClsComplianceRequired* method to determine if the given method is required to be CLS Compliant or not.
Thread Safe:	No
C# Method Signature:	`public override bool IsClsComplianceRequired(DeclSpace container);`
Version Information:	New in Version 0.2.5.1 of the SDK

IsExcluded

<table>
<tr><td>Name:</td><td>IsExcluded</td></tr>
<tr><td>Description:</td><td>Determines if the method has a conditional attribute whose conditions are not met.</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>EmitContext</td><td>The emit context which is being used to generate the actual MSIL code.</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The IsExcluded method is used to determine if a method has one or more conditional attributes attached to it. If such attributes are present, and the conditions are not defined, IsExcluded will return true. That is, if the method is to be excluded.

IsExcluded will return false if no such conditional attributes are present or the conditions which are specified in the conditional attributes are defined.

For more information on conditional attributes, please see the .NET Framework 2.0 SDK documentation [7].</td></tr>
<tr><td>C# Example:</td><td>public static bool IsMethodExcluded(Method method, EmitContext ec)
{
    return method.IsExcluded(ec);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method uses the IsExcluded method to determine if the given method contains any conditional attributes, whose conditions are not defined. The example method will return true if the given method contains such attibutes, and will be excluded during the MSIL code generation phase. The example method will return false otherwise.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public bool IsExcluded(EmitContext ec);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

IsExposedFromAssembly

<table>
<tr><td>Name:</td><td>IsExposedFromAssembly</td></tr>
<tr><td>Description:</td><td>Checks to see if this method is exposed from the assembly</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>DeclSpace</td><td>The containing type</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The IsExposedFromAssembly method performs checks to see if the method is exposed from the assembly or not. Being exposed from an assembly simply means that program elements that reside outside of the type method's assembly are able to view the method. That is, the method's accessibility modifiers allow it to be seen outside of the assembly.

The "DeclSpace" parameter is used to specify the containing type which contains the method.

The IsExposedFromAssembly will return true if the method is visible from outside the assembly, false otherwise.</td></tr>
<tr><td>C# Example:</td><td>public static bool IsTypeContainerExposed(Method method)
{
    return method.IsExposedFromAssembly(method.Parent);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method uses the IsTypeContainerExposed method to determine if the given method is visible from outside the assembly or not.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public bool IsExposedFromAssembly(DeclSpace ds);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

SetMemberIsUsed

Name:	**SetMemberIsUsed**
Description:	**Marks the method as being used (called)**
Parameters:	o
Parameter Values:	N/A
Remarks:	The *SetMemberIsUsed* method sets an internal flag in the method to mark it as used within the current code base. Normally, a program element's usage is calculated automatically by the C# compiler. However, sometimes when adding code via a static extension, the compiler may indicate that a given program element is not used. This method will allow you to remove that warning from the compiler output.

C# Example:	public static void MarkAsUsed(Method method)
	{
	method.SetMemberIsUsed();
	}

Example Notes:	The example method uses the *SetMemberIsUsed* method to mark the given method as used.
Thread Safe:	Yes
C# Method Signature:	public void SetMemberIsUsed();
Version Information:	New in Version 0.2.5.1 of the SDK

SetYields

Name:	**SetYields**
Description:	**Marks the method as containing the yields modifier**
Parameters:	0
Parameter Values:	N/A

Remarks:	The *SetYields* method updates the method's modifier flags to include the yields modifier. The *SetYields* method should be invoked if you add a yield statement to either the method's top level block or any sub block.
	
	The "yield" statement is used in an iterator block to provide a value to the enumerator object or to signal the end of an iteration. The "yield" statement can only appear inside an iterator block, which might be used as a body of a method, operator, or accessor. The following is a short example of the "yield" statement:
	
	`using System;`
	`using System.Collections;`
	`public class List`
	`{`
	`  public static IEnumerable Power(int number, int exponent)`
	`  {`
	`  int counter = 0;`
	`  int result = 1;`
	`  while (counter++ < exponent)`
	`  {`
	`  result = result * number;`
	`  yield return result;`
	`  }`
	`  }`
	
	`  static void Main()`
	`  {`
	`  // Display powers of 2 up to the exponent 8:`
	`  foreach (int i in Power(2, 8))`
	`  {`
	`  Console.Write("{0} ", i);`
	`  }`
	`  }`
	`}`
	
	For more information on the "yield" statement, please see the .NET Framework 2.0 SDK documentation [7].

C# Example:	public static void MarkUsingYields(Method method)
	{
	method.SetYields();
	}

Example Notes:	The example method uses the *SetYields* method to mark the given method with the yields modifier. The yields modifier indicates that the "yield" statement has been inserted in one of the blocks that compose the method's body.
Thread Safe:	Yes
C# Method Signature:	public void SetYields();
Version Information:	New in Version 0.2.5.1 of the SDK

UnsafeOK

<table>
<tr><td>Name:</td><td>UnsafeOK</td></tr>
<tr><td>Description:</td><td>Checks to see if this method can use pointers</td></tr>
<tr><td>Parameters:</td><td>1</td></tr>
<tr><td>Parameter Values:</td><td>
<table>
<tr><th>Ordinal</th><th>Type</th><th>Description</th></tr>
<tr><td>1</td><td>DeclSpace</td><td>The containing type</td></tr>
</table>
</td></tr>
<tr><td>Remarks:</td><td>The UnsafeOK method performs checks to see if it is okay to use an unsafe pointer within the method.

The “DeclSpace” parameter is used to specify the containing type which contains the method.

The UnsafeOK method will return true if the method can use unsafe pointers, false otherwise.</td></tr>
<tr><td>C# Example:</td><td>public static bool CanUsePointers(Method method)
{
    return method.UnsafeOK(method.Parent);
}</td></tr>
<tr><td>Example Notes:</td><td>The example method uses the UnsafeOK method to determine if the given method can use unsafe pointers or not.</td></tr>
<tr><td>Thread Safe:</td><td>No</td></tr>
<tr><td>C# Method Signature:</td><td>public bool UnsafeOK(DeclSpace parent);</td></tr>
<tr><td>Version Information:</td><td>New in Version 0.2.5.1 of the SDK</td></tr>
</table>

Properties

The following tables will discuss each of the properties defined by the Method type. These properties can be used to access the structural elements contained within the Method.

AttributeTargets

Name:	**AttributeTargets**
Description:	**Gets the type of attributes which can be applied to the method**
Read Only:	Yes
C# Signature:	public AttributeTargets AttributeTargets
Remarks:	The *AttributeTargets* property returns a System.AttributeTargets enumeration value indicating the type of attributes which can be applied to this method. The only possible return value is as follows: AttributeTargets.Method;

C# Example:	public static AttributeTargets GetAttributeTargets(Method method)
	{
	return method.AttributeTargets;
	}

Example Notes:	Uses the property to get the type of attributes which can be applied to the given method.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Block

Name:	**Block**
Description:	**Gets the top level block which defines the body of the method**
Read Only:	No
C# Signature:	ToplevelBlock Block
Remarks:	The *Block* property returns a ToplevelBlock object which represents the body of the method. If the method is abstract or part of an interface, the B*lock* property will contain a null value. Blocks, top level or otherwise, consist of a list of statements which make up the body of a given method. When performing static checks it can be difficult to iterate through all of the statements contained within a given block, especially when only a single type of statement is requested. To this end, the Contract Evaluation Engine Extension SDK contains a DeepVisitor class which examines blocks and calls a dedicated method for each type of statement and expression encountered in the block. The DeepVisitor class evaluates all code paths defined within a method. That is, all sub-statements and control flow branches and their sub-statements defined within a given method are visited. An example of how to statically analyze a method body, via its statement block will be provided later in this document. Developers can use the *Block* property to access and modify a method's behavior. As the *Block* property is not read-only, entire method blocks can be substituted.

C# Example:	public static int GetStatementCount(Method method)
	{
	if (method.Block == null) return 0;
	return method.Block.Statements.Count;
	}

Example Notes:	The example method returns the number of statements contained within the given method's body. The number of statements is determined by accessing the "Statements" property defined in the "Block" class. The *Block* property is used to access the given method's top level block.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

CallingConventions

Name:	**CallingConventions**
Description:	**Gets the calling conventions supported by the method**
Read Only:	Yes
C# Signature:	public CallingConventions CallingConventions
Remarks:	The *CallingConventions* property returns a System.Reflection.CallingConventions enumeration value indicating the valid calling conventions for the method. Valid calling conventions are: CallingConventions.Any; CallingConventions.ExplicitThis; CallingConventions.HasThis; CallingConventions.Standard; CallingConventions.VarArgs; The native calling convention is the set of rules governing the order and layout of arguments passed to compiled methods. It also governs how to pass the return value, what registers to use for arguments, and whether the called on the calling method removes arguments from the stack.

C# Example:	public static bool HasVarArgsConvention(Method method)
	{
	return ((method.CallingConventions &
	CallingConventions.VarArgs) != 0);
	}

Example Notes:	Uses the property to determine the calling conventions for the given method. The example method returns true if the given method contains the "VarArgs" calling convention, false otherwise.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

DocCommentHeader

Name:	**DocCommentHeader**
Description:	**Gets the xml document header string for xml comment output**
Read Only:	Yes
C# Signature:	public override string DocCommentHeader
Remarks:	The *DocCommentHeader* property returns the xml document header string for xml comment output. This property is only used for xml comment generation. It is included in the Contract Evaluation Engine Extension SDK so that if xml code comments are being generated as part of static evaluation the necessary API is available.

C# Example:	public static string GetDocCommentName(Method method)
	{
	return String.Concat(method.DocCommentHeader,
	method.Name);
	}

Example Notes:	The example code returns the correct xml comment name for the given method.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

InUnsafe

Name:	**InUnsafe**
Description:	**Determines if this method is marked as unsafe**
Read Only:	Yes
C# Signature:	public bool InUnsafe
Remarks:	The *InUnsafe* property indicates if the given method is marked as unsafe. Unsafe methods are able to use pointers. A method is marked as unsafe by using the "unsafe" keyword as a modifier when the method is defined.

C# Example:	public static bool IsMethodUnsafe(Method method)
	{
	return method.InUnsafe;
	}

Example Notes:	The example code uses the *InUnsafe* property to determine if the given method is marked as unsafe or not.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

IsUsed

Name:	**IsUsed**
Description:	**Determines if this method is used or not**
Read Only:	Yes
C# Signature:	public virtual bool IsUsed
Remarks:	The *IsUsed* property indicates if the given method is used or referenced by another program element. The property will return true if the method is used by another program element, false otherwise. Developers can mark a method as used via the "SetMemberIsUsed" method. The method has been previously discussed.

C# Example:	public static bool IsMethodUsed(Method method)
	{
	  return method.IsUsed;
	}

Example Notes:	The example code uses the *IsUsed* property to determine if the given method is used or referenced by another program element.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Location

Name:	**Location**
Description:	**Gets the source code location where the method was defined**
Read Only:	Yes
C# Signature:	public Location Location
Remarks:	The *Location* property represents the source location where the method was defined. That is, the source code file name and line number. The line number specified is where the first line of the method's definition is located. The "Location" class contains several members to access the source code file, line number, and column where the method's definition is located.

C# Example:	public static int GetMethodDefinitionLineNumber(Method method)
	{
	return method.Location.Row;
	}

Example Notes:	The example code uses the *Location* property to access the source code location where the method was defined. Once the location information is retrieved, the example code returns the line number (row) where the definition is located.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

MemberName

Name:	**MemberName**
Description:	**Gets the member name object that represents the method**
Read Only:	Yes
C# Signature:	public MemberName MemberName
Remarks:	The *MemberName* property returns a member name object which represents the name of the method. The "MemberName" type contains the name, namespace, and source code location where the method has been defined. Each of these elements can be accessed via dedicated properties defined on the "MemberName" type. If you are only interested in the name of the method, use the "Name" property instead.

C# Example:	public static string GetFullName(Method method)
	{
	return method.MemberName.GetName();
	}

Example Notes:	The example code uses the *GetMemberName* property to get the member name object, and then calls the "GetName" method defined on the "MemberName" type. The "GetName" method returns a string representation of the fully resolved name of the given method.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

MemberType

Name:	**MemberType**
Description:	**Gets the method's type**
Read Only:	Yes
C# Signature:	public Type MemberType
Remarks:	The *MemberType* property returns a System.Type object that represents method's type. As regular methods do not define or use one specific type, the *MemberType* property will contain a null value. However, in the case of properties, indexers, or events, the *MemberType* property will return the System.Type object that represents the type of property, indexer, or event. If the method's type has not yet been resolved, the property will resolve the method's type. If such a resolution cannot be performed, the property will return null. For more information on the System.Type class, please see the .NET Framework 2.0 documentation [7].

C# Example:	public static Type GetMethodType(Method method)
	{
	return method.MemberType;
	}

Example Notes:	The example code uses the *MemberType* property to get the System.Type object that represents the method's type.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

MethodName

Name:	**MethodName**
Description:	**Gets a MemberName object that represents the method's name.**
Read Only:	Yes
C# Signature:	public MemberName MethodName
Remarks:	The *MethodName* property returns a member name object which represents the name of the method. The "MemberName" type contains the name, parameter types, return type, namespace, and source code location where the method has been defined. Each of these elements can be accessed via dedicated properties defined on the "MemberName" type. If you are only interested in the name of the method, use the "Name" property instead.

C# Example:	public static string GetFullName(Method method)
	{
	return method.MethodName.GetName();
	}

Example Notes:	The example code uses the *MethodName* property to get the member name object, and then calls the "GetName" method defined on the "MemberName" type. The "GetName" method returns a string representation of the fully resolved name of the given method.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Name

Name:	**Name**
Description:	**Gets the name of this method**
Read Only:	Yes
C# Signature:	public string Name
Remarks:	The *Name* property returns a string representation of the method's name. This name will include namespace information as well as the method's actual name. The *Name* property yields the same result as calling "MemberName.GetName(false)" however, the name is cached so the *Name* property should be used instead.

C# Example:	public static string GetMethodName(Method method)
	{
	return method.Name;
	}

Example Notes:	The example code uses the *Name* property to get the full name of the given method.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

OptAttributes

Name:	**OptAttributes**
Description:	**Contains a list of all the attributes which have been applied to this method**
Read Only:	Yes
C# Signature:	public Attributes OptAttributes
Remarks:	The *OptAttributes* property contains a list of all the attributes which have been applied to this method.

C# Example:	public static bool HasAttributes(Method method)
	{
	return (method.OptAttributes != null);
	}

Example Notes:	The example code examines the given method to see if the method contains any optional attributes. If the method contains one or more optional attributes, the example returns true, false otherwise.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

ParameterInfo

Name:	**ParameterInfo**
Description:	**Gets the list of parameters which the method accepts**
Read Only:	Yes
C# Signature:	Parameters ParameterInfo
Remarks:	The *ParameterInfo* property provides access to the set of parameters that the method accepts. The parameter list is represented via the "Parameters" class. The Parameters class provides a wrapper to the underlying parameter set. Parameters can be accessed via index or by parameter name. If a method does not contain any parameters, the *ParameterInfo* property will still return a valid "Parameters" object, however the object's "Empty" property will return true.

C# Example:	public static int ParameterCount(Method method)
	{
	if (method.ParameterInfo.Empty) return 0;
	return method.ParameterInfo.Count;
	}

Example Notes:	The example code returns the number of parameters that the given method accepts. The code first checks to see if the *ParameterInfo* property contains an empty parameter set, in which case a value of zero is returned. If the *ParameterInfo* property contains a non zero number of parameters, then the number of parameters is returned to the caller, via the "Count" property defined on the "Parameters" type.
Thread Safe:	Methods – Yes, Properties, Indexers, and Constructors - No
Version Information:	New in Version 0.2.5.1 of the SDK

ParameterTypes

Name:	**ParameterTypes**
Description:	**Gets and array of types for the method's parameters**
Read Only:	Yes
C# Signature:	public Type[] ParameterTypes
Remarks:	The *ParameterTypes* property provides access to the set of parameters that the method accepts. The type array contains a System.Type object for each parameter the method accepts (in order). If a method does not contain any parameters, the *ParameterTypes* property will return an empty array.

C# Example:	public static Type GetFirstParameterType(Method method)
	{
	if (method.ParameterTypes.Length == 0) return null;
	return method.ParameterTypes[0];
	}

Example Notes:	The example code returns a System.Type object which represents the type of the first parameter that is defined by the given method. If the given method does not accept any parameters, then a value of null is returned.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

Parent

Name:	**Parent**
Description:	**Contains the type container where the method has been defined**
Read Only:	Yes
C# Signature:	public new TypeContainer Parent
Remarks:	The *Parent* property contains the type container where the method has been defined. This value cannot be null, as C# does not permit global methods.

C# Example:	public static TypeContainer GetMethodContainer(Method method)
	{
	  return method.Parent;
	}

Example Notes:	The example code gets the TypeContainer object that contains the given method. That is, the type container where the method is defined is returned.
Thread Safe:	No
Version Information:	New in Version 0.2.5.1 of the SDK

ReturnType

Name:	**ReturnType**
Description:	**Gets a System.Type object that represents the method's return type**
Read Only:	Yes
C# Signature:	public Type ReturnType
Remarks:	The *ReturnType* property returns a System.Type object that represents the return type of the method. If the method does not contain a return type, that is has a "void" return type. The *ReturnType* property will return null. For more information on the System.Type class, please see the .NET Framework 2.0 SDK documentation [7].

C# Example:	public static bool IsIntReturnType(Method method)
	{
	return (method.ReturnType == typeof(int));
	}

Example Notes:	The example code uses the *ReturnType* property to get a System.Type object of the given method's return type. The example method will return true if the given method returns an integer. The example method will return false otherwise.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

ShortName

Name:	**ShortName**
Description:	**Contains a string representation of the method's name without the namespace or explicit interface definition name**
Read Only:	No
C# Signature:	public string ShortName
Remarks:	The *ShortName* property contains a string representation of the method's name. This name does not contain any namespace or containing type information. In addition, the name does not contain the explicit interface definition name, if one exists. The *ShortName* property can be written to. This will effectively rename the method. It will not modify any code which calls the method.

C# Example:	public static void RenameMethod(Method method, string newName)
	{
	method.ShortName = newName;
	}

Example Notes:	The example code uses the *ShortName* property to rename the given method. This code will not modify any body code that may make reference to the method.
Thread Safe:	Read – Yes, Write – No
Version Information:	New in Version 0.2.5.1 of the SDK

ValidAttributeTargets

Name:	**ValidAttributeTargets**
Description:	**Returns a string array of valid attribute targets for the given method**
Read Only:	Yes
C# Signature:	public override string[] ValidAttributeTargets
Remarks:	The *ValidAttributeTargets* property returns an array which indicates the type of attributes that can be applied to the method. For methods the resultant array will contain two strings, "method" and "return".

C# Example:	public static bool IsMethodAttributeTarget(Method method)
	{
	string[] targets = method.ValidAttributeTargets;
	if (targets == null)
	return false;
	foreach (string target in targets)
	if (target == "method")
	return true;
	return false;
	}

Example Notes:	The example code uses the *ValidAttributeTargets* property to get the list of valid attribute targets, and then checks the list to see if one of the attribute targets is a "method". If a method is found the example method will return true. If a "method" target is not found the example method will return false.
Thread Safe:	Yes
Version Information:	New in Version 0.2.5.1 of the SDK

Attributes

The following tables will discuss each of the public attributes defined by the Method type. These attributes can be used to access the elements contained within the method as well as information about the method itself.

DocComment

Name:	**DocComment**
Description:	**A string representation of the XML documentation comment**
C# Signature:	public string DocComment;
Remarks:	The *DocComment* attribute contains a string representation of the XML documentation comment which preceded the method's definition. If no such comment was specified, the *DocComment* attribute will be set to an empty string.
Version Information:	New in Version 0.2.5.1 of the SDK

flags

Name:	**flags**
Description:	**Specifies any additional flags that represent method attributes**
C# Signature:	public MethodAttributes flags;
Remarks:	The *flags* attribute specifies flags for the method's attributes. The MethodAttributes type is an enumeration that allows for bitwise combination of its values. Possible values are as follows:

Member name	Description
Abstract	Indicates that the class does not provide an implementation of this method.
Assembly	Indicates that the method is accessible to any class of this assembly.
CheckAccessOnOverride	Indicates that the method can only be overridden when it is also accessible.
FamANDAssem	Indicates that the method is accessible to members of this type and its derived types that are in this assembly only.
Family	Indicates that the method is accessible only to members of this class and its derived classes.
FamORAssem	Indicates that the method is accessible to derived classes anywhere, as well as to any class in the assembly.

	Final	Indicates that the method cannot be overridden.
	HasSecurity	Indicates that the method has security associated with it. Reserved flag for runtime use only.
	HideBySig	Indicates that the method hides by name and signature; otherwise, by name only.
	MemberAccessMask	Retrieves accessibility information.
	NewSlot	Indicates that the method always gets a new slot in the vtable.
	PinvokeImpl	Indicates that the method implementation is forwarded through PInvoke (Platform Invocation Services).
	Private	Indicates that the method is accessible only to the current class.
	PrivateScope	Indicates that the member cannot be referenced.
	Public	Indicates that the method is accessible to any object for which this object is in scope.
	RequireSecObject	Indicates that the method calls another method containing security code. Reserved flag for runtime use only.
	ReservedMask	Indicates a reserved flag for runtime use only.
	ReuseSlot	Indicates that the method will reuse an existing slot in the vtable. This is the default behavior.
	RTSpecialName	Indicates that the common language runtime checks the name encoding.
	SpecialName	Indicates that the method is special. The name describes how this method is special.

	Static	Indicates that the method is defined on the type; otherwise, it is defined per instance.
	UnmanagedExport	Indicates that the managed method is exported by thunk to unmanaged code.
	Virtual	Indicates that the method is virtual.
	VtableLayoutMask	Retrieves vtable attributes.
	For more information, please see the .NET Framework 2.0 Software Development Kit (SDK) [7].	
Version Information:	New in Version 0.2.5.1 of the SDK	

`InterfaceType`

Name:	**InterfaceType**
Description:	**Indicates the interface type that the method explicitly implements**
C# Signature:	public Type InterfaceType = null;
Remarks:	The *InterfaceType* attribute indicates the interface type that the method is explicitly implementing. The interface type is represented by a System.Type object. If the method does not explicitly implement any interface, the *InterfaceType* attribute will always contain its initial value of null.
Version Information:	New in Version 0.2.5.1 of the SDK

`IsExplicitImpl`

Name:	**IsExplicitImpl**
Description:	**Indicates if this method explicitly implements an interface or not**
C# Signature:	public bool IsExplicitImpl;
Remarks:	The *IsExplicitImpl* attribute will contain a value of true, if this method explicitly implements and interface. The type of interface can be acquired via the *InterfaceType* attribute. If the method does not implement such an interface, the *IsExplicitImpl* field will contain a value of false.
Version Information:	New in Version 0.2.5.1 of the SDK

IsInterface

Name:	**IsInterface**
Description:	**Indicates if this method is an interface member**
C# Signature:	public bool IsInterface;
Remarks:	The *IsInterface* field will contain a value of true, if this method is an interface member. That is, the method does not have a body defined, and the *ToplevelBlock* property will contain a null block. If the *IsInterface* field contains a false value, the method is not an interface member and will contain a valid top level block.
Version Information:	New in Version 0.2.5.1 of the SDK

IsOperator

Name:	**IsOperator**
Description:	**Indicates if the method represents an operator method**
C# Signature:	public Operator IsOperator;
Remarks:	The *IsOperator* attribute will contain a non-null value if the method represents an operator method. The Operator object represented by the *IsOperator* contains the operator's information, including operator type, and the operator's code block.
Version Information:	New in Version 0.2.5.1 of the SDK

MethodBuilder

Name:	**MethodBuilder**
Description:	**Points to the actual method definition that is being created**
C# Signature:	public MethodBuilder MethodBuilder;
Remarks:	The *MethodBuilder* attribute contains a reference to the actual method definition that is being created by the method. The MethodBuilder is part of the System.Reflection.Emit namespace. The method builder will be used when generating the actual MSIL code when re-assembling the SUT. Developers can use the *MethodBuilder* field to work directly with the code generation process via the types defined within the System.Reflection.Emit namespace. For more information please see the .NET Framework 2.0 Software Development Kit (SDK) [7].
Version Information:	New in Version 0.2.5.1 of the SDK

MethodData

Name:	**MethodData**
Description:	**Additional data used by the method**
C# Signature:	public MethodData MethodData;
Remarks:	The *MethodData* attribute represents a container class, which encapsulates most of the method's state. Almost all of the data contained within the *MethodData* attribute is accessible via the methods previously discussed. The remaining data does not have a use when creating static extensions. Developers can use the *MethodData* field to work directly with the method's internal state and behaviour.
Version Information:	New in Version 0.2.5.1 of the SDK

ModFlags

Name:	**ModFlags**
Description:	**Indicates which modifier flags have been applied to the method**
C# Signature:	public int ModFlags;

Remarks:	The *ModFlags* attribute contains any modifier flags that have been applied to the method. To determine if a given modifier flag has been applied to the method, use an and (&) bit mask, against one of the following values:
	
	public const int PROTECTED = 0x0001;
	public const int PUBLIC = 0x0002;
	public const int PRIVATE = 0x0004;
	public const int INTERNAL = 0x0008;
	public const int NEW = 0x0010;
	public const int ABSTRACT = 0x0020;
	public const int SEALED = 0x0040;
	public const int STATIC = 0x0080;
	public const int READONLY = 0x0100;
	public const int VIRTUAL = 0x0200;
	public const int OVERRIDE = 0x0400;
	public const int EXTERN = 0x0800;
	public const int VOLATILE = 0x1000;
	public const int UNSAFE = 0x2000;
	public const int TOP = 0x2000;
	public const int PROPERTY_CUSTOM = 0x4000;
	public const int METHOD_YIELDS = 0x8000;
	public const int METHOD_GENERIC = 0x10000;
	
	The values listed above are defined in the .NET Framework 2.0 Software Development Kit [7].

Version Information:	New in Version 0.2.5.1 of the SDK

Parameters

Name:	**Parameters**
Description:	**Provides direct access to the method's parameter set**
C# Signature:	public readonly Parameters Parameters;
Remarks:	The *Parameters* attribute provides access to the set of parameters that the method accepts. The parameter list is represented via the "Parameters" class. The Parameters class provides a wrapper to the underlying parameter set. Parameters can be accessed via index or by parameter name. If a method does not contain any parameters, the *Parameters* attribute will still return a valid "Parameters" object, however the object's "Empty" property will return true.
Version Information:	New in Version 0.2.5.1 of the SDK

Parent

Name:	**Parent**
Description:	**Contains the parent of the method**
C# Signature:	public DeclSpace Parent;
Remarks:	The *Parent* attribute contains the parent of this method. For all methods, the parents will be the type container where the method resides.
Version Information:	New in Version 0.2.5.1 of the SDK

Type

Name:	**Type**
Description:	**Contains the return type of this method**
C# Signature:	public Expression Type;
Remarks:	The *Type* field contains the type expression that represents the return type for the method. If the method being defined is property, event, or indexer then the type expression contains the type of the property, event, or indexer. A System.Type object can be retrieved by using the *MemberType* property.
Version Information:	New in Version 0.2.5.1 of the SDK

Real

The Real type represents a double precision floating point value. Real values are signed, and are 64-bits in length. They have a range of ±5.0 × 10−324 to ±1.7 × 10308. To specify a static entry point that takes a parameter of the Real type, use the .NET System.Double structure. This structure is aliased by the *double* keyword in most .NET languages. Figure 13, illustrates a static extension that takes one parameter of the Real type.

```
[StaticEntryPoint]
public StaticEvaluationResult DoEvaluation(TypeContainer type, double dValue)
{
        // To Do: Put code to implement the static extension here
}
```

Figure 13 – Static Entry Point Signature with a Real Parameter

String

The String type represents text as a series of Unicode characters. To specify a static entry point that takes a parameter of the String type, use the .NET System.String class. This structure is aliased by the *string* keyword in most .NET languages. Figure 14, illustrates a static extension that takes one parameter of the String type.

```
[StaticEntryPoint]
public StaticEvaluationResult DoEvaluation(TypeContainer type, string sValue)
{
        // To Do: Put code to implement the static extension here
}
```

Figure 14 – Static Entry Point Signature with a String Parameter

Analyzing Method Behaviour

As we have discussed in the previous sections, using the various parameter types and their corresponding methods, properties, and attributes we can statically analyze and modify any structural element contained within the SUT. However, with the exception of accessing a method's code block via the *ToplevelBlock* property defined on the Method parameter type, we have not discussed how to directly examine a method's behaviour and control flow. The following subsections will look at how such behaviour and control flow can be examined.

Each type of statement and expression that can be used to define all or part of a method's behaviour is represented by a dedicated class. Each of these classes is located in the "DaveArnold.Contracts.CSCompiler" namespace and is named after the statement or expression that they represent. For example, the Switch class represents the C# "switch" statement. Each of the dedicated classes has members, which can be used to access the sub-expression and statement which comprise the original statement.

Behavioural Visitors

Normally, during the execution of a given static check we only want to search for a certain type of statement or expression. For example, maybe we want to ensure that a given method body contain a switch statement which operates on a specific variable. In this case, we really don't want to have to go through each and every type of statement and expression until we find the switch. In order to aid with such statement and expression analysis, the Contract Evaluation Engine Extension SDK, contains two visitor interfaces. The visitor interfaces are located in the "DaveArnold.Contracts.CSCompiler.ContractExtensions" namespace. One of the interfaces, IStatementVisitor, contains a "ProcessStatement" method for each type of statement that can be encountered within a method block. The second interface, IExpressionVisitor, contains a "ProcessExpression" method for each type of expression that can be encountered within a method block. Figures 15 and 16 provide code listings for the two visitor interfaces respectively.

```
namespace DaveArnold.Contracts.CSCompiler.ContractExtensions
{
        public interface IStatementVisitor
        {
                void ProcessStatement(Unsafe stmt);
                void ProcessStatement(Lock stmt);
                void ProcessStatement(Break stmt);
                void ProcessStatement(Block stmt);
                void ProcessStatement(Catch stmt);
                void ProcessStatement(Iterator.MoveNextStatement stmt);
                void ProcessStatement(Iterator.DisposeMethod stmt);
                void ProcessStatement(Iterator.StatementList stmt);
                void ProcessStatement(Iterator.SetState stmt);
                void ProcessStatement(Iterator.InitScope stmt);
                void ProcessStatement(Iterator.NoCheckReturn stmt);
                void ProcessStatement(Try stmt);
                void ProcessStatement(For stmt);
                void ProcessStatement(EmptyStatement stmt);
                void ProcessStatement(Continue stmt);
                void ProcessStatement(GotoCase stmt);
                void ProcessStatement(Foreach stmt);
                void ProcessStatement(Foreach.ArrayForeach stmt);
                void ProcessStatement(Foreach.CollectionForeach stmt);
                void ProcessStatement(Foreach.CollectionForeachStatement stmt);
                void ProcessStatement(Throw stmt);
                void ProcessStatement(StatementExpression stmt);
                void ProcessStatement(Return stmt);
                void ProcessStatement(YieldBreak stmt);
                void ProcessStatement(Using stmt);
                void ProcessStatement(LabeledStatement stmt);
                void ProcessStatement(Goto stmt);
                void ProcessStatement(Fixed stmt);
                void ProcessStatement(If stmt);
                void ProcessStatement(GotoDefault stmt);
                void ProcessStatement(Unchecked stmt);
                void ProcessStatement(Switch stmt);
                void ProcessStatement(While stmt);
                void ProcessStatement(Do stmt);
                void ProcessStatement(Yield stmt);
                void ProcessStatement(Checked stmt);
        }
}
```

Figure 15 – The IStatementVisitor Interface

```
namespace DaveArnold.Contracts.CSCompiler.ContractExtensions
{
        public interface IExpressionVisitor
        {
                void ProcessExpresson(TypeExpression exp);
                void ProcessExpresson(MethodGroupExpr exp);
                void ProcessExpression(InvocationOrCast exp);
                void ProcessExpression(EmptyExpression exp);
                void ProcessExpression(EmptyCast exp);
                void ProcessExpression(Unary exp);
                void ProcessExpression(StringConcat exp);
                void ProcessExpression(ProxyInstance exp);
                void ProcessExpression(ULongConstant exp);
                void ProcessExpression(Iterator.SimpleParameterReference exp);
                void ProcessExpression(Iterator.CapturedParameterReference exp);
                void ProcessExpression(Iterator.CapturedThisReference exp);
                void ProcessExpression(Iterator.FieldExpression exp);
                void ProcessExpression(Iterator.MoveNextMethod exp);
                void ProcessExpression(PropertyExpr exp);
                void ProcessExpression(IndexerAccess exp);
                void ProcessExpression(BinaryDelegate exp);
                void ProcessExpression(StringConstant exp);
                void ProcessExpression(StackAlloc exp);
                void ProcessExpression(DoubleConstant exp);
                void ProcessExpression(SizeOf exp);
                void ProcessExpression(ParameterReference exp);
                void ProcessExpression(ElementAccess exp);
                void ProcessExpression(UserCast exp);
                void ProcessExpression(DelegateInvocation exp);
                void ProcessExpression(AnonymousDelegate exp);
                void ProcessExpression(
                        AnonymousDelegate.AnonymousInstance exp);
                void ProcessExpression(TypeOf exp);
                void ProcessExpression(TypeLookupExpression exp);
                void ProcessExpression(Invocation exp);
                void ProcessExpression(ImplicitDelegateCreation exp);
                void ProcessExpression(EnumConstant exp);
                void ProcessExpression(TemporaryVariable exp);
                void ProcessExpression(FloatConstant exp);
                void ProcessExpression(UnaryMutator exp);
                void ProcessExpression(TypeAliasExpression exp);
                void ProcessExpression(ParenthesizedExpression exp);
                void ProcessExpression(LocalTemporary exp);
                void ProcessExpression(DecimalConstant exp);
                void ProcessExpression(NewDelegate exp);
                void ProcessExpression(LocalVariableReference exp);
                void ProcessExpression(Binary exp);
                void ProcessExpression(ArrayCreation exp);
```

```
                void ProcessExpression(NullLiteral exp);
                void ProcessExpression(UnCheckedExpr exp);
                void ProcessExpression(FixedBufferPtr exp);
                void ProcessExpression(FieldExpr exp);
                void ProcessExpression(As exp);
                void ProcessExpression(ArglistAccess exp);
                void ProcessExpression(Arglist exp);
                void ProcessExpression(Namespace exp);
                void ProcessExpression(LongConstant exp);
                void ProcessExpression(CharConstant exp);
                void ProcessExpression(StaticCallExpr exp);
                void ProcessExpression(NullCast exp);
                void ProcessExpression(EventExpr exp);
                void ProcessExpression(CheckedExpr exp);
                void ProcessExpression(SByteConstant exp);
                void ProcessExpression(IntConstant exp);
                void ProcessExpression(StringPtr exp);
                void ProcessExpression(MemberAccess exp);
                void ProcessExpression(UIntConstant exp);
                void ProcessExpression(BoolConstant exp);
                void ProcessExpression(This exp);
                void ProcessExpression(UShortConstant exp);
                void ProcessExpression(ShortConstant exp);
                void ProcessExpression(Assign exp);
                void ProcessExpression(AnonymousMethod exp);
                void ProcessExpression(SimpleName exp);
                void ProcessExpression(Conditional exp);
                void ProcessExpression(BinaryMethod exp);
                void ProcessExpression(ByteConstant exp);
                void ProcessExpression(PointerArithmetic exp);
                void ProcessExpression(ConditionalLogicalOperator exp);
                void ProcessExpression(QualifiedAliasMember exp);
                void ProcessExpression(ArrayAccess exp);
                void ProcessExpression(New exp);
                void ProcessExpression(ComposedCast exp);
                void ProcessExpression(Cast exp);
                void ProcessExpression(Is exp);
                void ProcessExpression(Indirection exp);
                void ProcessExpression(BaseAccess exp);
                void ProcessExpression(TypeExpression exp);
                void ProcessExpression(MethodGroupExpr exp);
        }
}
```

Figure 16 – The IExpressionVisitor Interface

The Deep Visitor

The two visitor interfaces shown in Figures 15 and 16, are helpful but they do not contain the required flexibility to perform deep visitation operations, which would be required to obtain flow control and branching information for a given method. The Contract Evaluation Engine SDK contains a "DeepVisitor" abstract class which implements both the IStatementVisitor interface and the IExpressionVisitor interface to provide such functionality. The "DeepVisitor" class is located in the "DaveArnold.Contracts.ExtensionAPI" namespace. The "DeepVisitor" contains a single constructor, which takes an instance of a ToplevelBlock. The developer can then inherit from the "DeepVisitor" class and override only the methods that involve the statements or expressions that the developer is interested in. To begin the visitor operations, simply call the Go() method defined in the "DeepVisitor" class. The second static extension tutorial will present an example of how to inherit and use the "DeepVisitor" class.

Static Extension Tutorials

The following two sections will each provide a detailed step-by-step tutorial of how to create a static extension. Each of the static extensions will use some of the API and techniques previously discussed in this document. In order to implement the extensions, you will need a copy of the Contract Evaluation Engine Extension SDK, which is included with this document. You will also need Visual Studio .NET 2005 for compilation.

The first tutorial, presents a very simple static extension that checks for inheritance. The second tutorial uses the "DeepVisitor" class, to locate a switch statement that operates on a given field.

Tutorial 1 – InheritFrom

While a single Visual Studio project can be used to contain multiple extensions of various types, for ease of understanding we will create a separate Visual Studio project for each of our tutorials. Tutorial 1, will be implemented using the C# language.

Step 1 – Create Project

To begin create a new Visual C# Class Library Project (.dll), and call it InheritFrom, as shown in Figure 17.

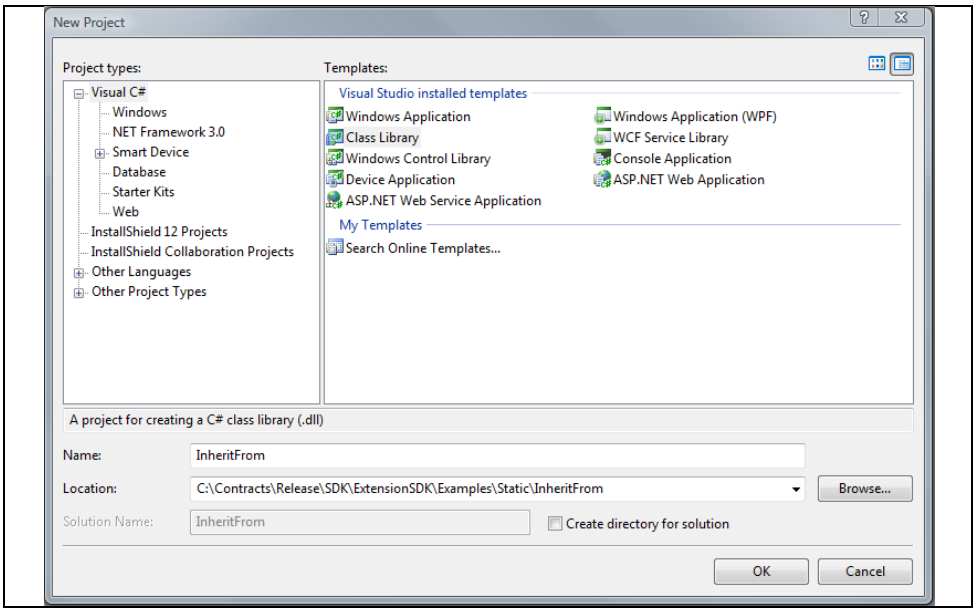


Figure 17 – Creating a new project for InheritFrom

Step 2 – Add References

Once the project has been created, you will be shown an initial code window with a dummy class named Class1.cs. You may delete this class, by selecting it from the Solution Explorer window and pressing the delete key. Before we can begin to create our Static Extension, we need to first reference the Contract Evaluation Engine Extension SDK. To do this, select the "References" folder in the Solution Explorer window and then right click. Select "Add Reference…" from the context menu as shown in Figure 18.

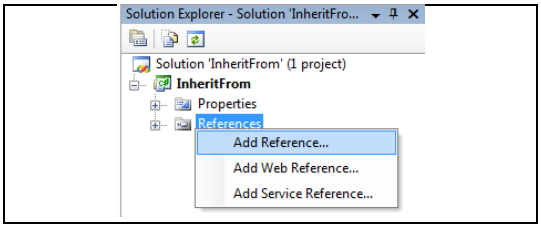


Figure 18 – Adding Contract Evaluation Engine SDK References

When the "Add Reference" dialog box is displayed, select the "Browse" tab, and in the folder where you have installed the Contract Evaluation Engine Extension SDK, select both the "CSCompiler.dll" and the "ExtensionAPI.dll" component files. When they are both selected, as shown

in Figure 19, click the "OK" button to add the required references. For the purposes of this example, the CSCompiler.dll file should have a file version of 1.1.13.8, and the ExtensionAPI.dll file should have a file version of 0.2.5.1.

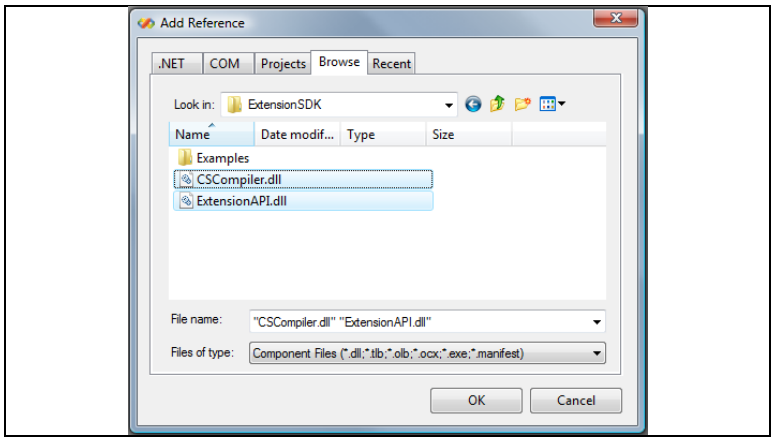


Figure 19 – Selecting SDK Components to Reference

Once the required SDK references has been added, the "References" folder located in the Solution Explorer should look like the one shown in Figure 20.

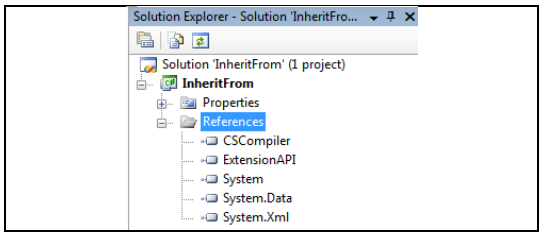


Figure 20 – SDK References Added Correctly

Step 3 – Basic Code Structure

Once the required references have been added to the project, add a new C# code file to the project, name the file "InheritFrom.cs". To begin the extension, enter the code shown in Figure 21, into your new code file.

```
using System;
using DaveArnold.Contracts.CSCompiler;
using DaveArnold.Contracts.ExtensionAPI;

namespace DaveArnold.Contracts.Extensions.Examples
{
        public class InheritFrom
        {
        }
}
```

Figure 21 – Initial InheritFrom C# Code

The first three lines import various namespaces into our code. The "System" namespace contains the core .NET types. The "CSCompiler" namespace contains the types, methods, properties, and attributes for the Class, Field, and Method parameter types already discussed. Finally, the "ExtensionAPI" namespace contains the generic classes required to implement a Contract Evaluation Engine Extension.

Following the using directives, a namespace is used to group the extensions together. There is no requirement for the extensions to reside in any particular namespace. Namespaces can be used for organization at the developers discretion, and have no effect on the naming of use of the extension itself. After the namespace, a simple public class is used to contain our extension. Any number of extensions may be present within one class, and the name of the class has no relationship to the name of the extension as you will soon see. It is good practice to name your class the same as the extension name, for code organization purposes, but the Contract Evaluation Engine Extension SDK does not use the class name for extension resolution. Finally, all extensions must reside within a class that is defined public. If the class does not contain the public modifier, the extension will not be recognized or loaded by the Contract Evaluation Engine.

Step 4 – Implement IExtension

In order for the extension to be recognized by the Contract Engine it must be located in a class that implements the IExtension interface. The IExtension interface is located in the "DaveArnold.Contracts.ExtensionAPI" which we have already imported, so all we need to do is change our class definition so it looks like this: "public class InheritFrom : IExtension". Make this change now.

When the IExtension interface is added to the class declaration, Visual Studio will mark the first letter of the interface name (in our case the "I"), with a blue underline. If you do not see a blue underline, simply click on the "I" in the "IExtension" interface name. If you click on the blue underline, you will get the menu shown in Figure 22.

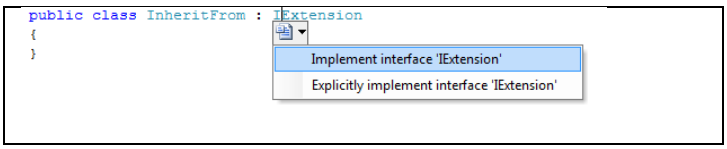


Figure 22 – Implementing the IExtension Interface

From the context menu that appears select the "Implement interface 'IExtension'" menu item. Visual Studio will now automatically insert members that will implement the IExtension interface into your new InheritFrom class. If everything was done correctly, your InheritFrom.cs should look like the one shown in Figure 23.

```
using System;
using DaveArnold.Contracts.CSCompiler;
using DaveArnold.Contracts.ExtensionAPI;

namespace DaveArnold.Contracts.Extensions.Examples
{
        public class InheritFrom : IExtension
        {
                #region IExtension Members

                public string Name
                {
                        get { throw new Exception(
                                "The method or operation is not
                                implemented."); }
                }
                public string Namespace
                {
                        get { throw new Exception(
                                "The method or operation is not
                                implemented."); }
                }
                public Version Version
                {
                        get { throw new Exception(
                                "The method or operation is not
                                implemented."); }
                }

                #endregion
        }
}
```

Figure 23 – InheritFrom C# Code after completing Step 4

Step 5 – Implement IExtension.Name

As shown in Figure 23, the IExtension interface only contains three read only properties that need to be implemented. The first is the Name property. As the name suggests, the Name property must return the name of the extension. This is the name that will be used to call the extension from the various high level contract programming languages, or directly from the CIL. As such, the name returned by the "Name" property must conform to a valid C++ method call. That is, the name cannot contain any spaces or punctuation symbols, and cannot begin with a number. Extension names may begin with underscores if desired. If an invalid extension name is provided, the extension will be ignored by the Contract Evaluation Engine.

To name our extension "CheckInheritance" replace the body of the Name property's get accessor with the code shown in Figure 24.

```
public string Name
{
        get { return "CheckInheritance"; }
}
```

Figure 24 – Naming the extension CheckInheritance

Step 6 – Implement IExtension.Namespace

The Namespace property is used to indicate where the extension resides. Extensions are organized into namespaces for ease of use and grouping. Extension namespaces have the same rules are C# namespaces. Each namespace cannot contain any spaces or punctuation characters. Namespaces are separated via the dot "." character.

In order to keep our simple tutorial extensions separate from the production ones, we will place our new extension in the "DaveArnold.Tutorials" namespace. To do this, replace the body of the Namespace property's get accessor with the code shown in Figure 25.

```
public string Namespace
{
        get { return "DaveArnold.Tutorials"; }
}
```

Figure 25 – Setting the Extension's Namespace to DaveArnold.Tutorials

Step 7 – Implement IExtension.Version

The final property to implement is the Version property. The Version property specifies the version number of the extension. Versions are represented by the System.Version structure, and consist of four numbers: major, minor, build, and revision numbers. The Version structure contains several constructors which allow you to set any number of the version components.

For the purposes of this tutorial, we will simply set the version of our extension to 1.0.0.0. To do this, replace the body of the Version property's get accessor with the code shown in Figure 26.

```
public Version Version
{
        get { return new Version(1, 0, 0, 0); }
}
```

Figure 26 – Setting the Extension's Version to 1.0.0.0

Step 8 – Define the Static Entry Point

Our extension now has all the required components for it to be a static extension. The problem is that, we have not defined the behaviour for our extension. To do this, we need to create a static entry point. As the name suggests, the entry point will act as our extension's main method. The entry point will be called each time the extension is called from within the Contract Evaluation Engine. To define a static entry point, add the code shown in Figure 27, to your InheritFrom.cs code file.

```
[StaticEntryPoint]
public StaticEvaluationResult DoStaticEvaluation(TypeContainer type, TypeContainer baseType)
{
}
```

Figure 27 – Defining a Static Entry Point

The static entry point shown in Figure 27, takes two parameters. The first parameter is the required context parameter. That is, it represents the current SUT type that the contract is evaluating on. The second parameter will represent the SUT type that we are going to check to see if the current SUT type derives from. That is, we are going to check to see if the type represented by the first parameter derives from the type represented by the second parameter.

Before we go ahead and implement the behaviour for our extension, a few things to note. The method name "DoStaticEvaluation" is completely arbitrary. The only required name is the use of the "StaticEntryPoint" attribute. It is the attribute which will alert the Contract Evaluation Engine where the entry point is located. All entry points must have their access modifiers set to public. If a non-public entry point is located, it will be ignored.

As previously discussed, all static entry points must have a return type of "StaticEvaluationResult", and must have at least one parameter of the "TypeContainer" type which will represent the current SUT type we are working with. However, any remaining parameters can be whatever is required in order to perform the desired evaluation as long as they conform to one of the parameter types previously discussed in this document.

The actual SUT types, methods, and fields which will be passed to an extension vary depending on the bindings that the test developer selects during the compilation of the high-level contract language. There is no way to access the binding tables from within any type of extension. Extensions, should function on any type, method, or field, and not assume that they will be passed a specific one each and every time, as binding information can change on every execution.

Step 9 – Implement the Static Entry Point

The only task remaining is to fill in the method body for the entry point that we created in Step 8. To do this, we will use some of the method and properties that we have already seen on the "Class" parameter type. Complete the method body as shown in Figure 28.

```
[StaticEntryPoint]
public StaticEvaluationResult DoStaticEvaluation(TypeContainer type, TypeContainer baseType)
{
        TypeExpr spare = null;
        TypeExpr[] bases = type.GetClassBases(out spare);
        if (bases != null && bases.Length != 0)
        {
                foreach (TypeExpr te in bases)
                {
                        if (te.Name == baseType.Name)
                                return new StaticEvaluationResult(StaticEvaluationResults.Pass);
                }
        }
        return new StaticEvaluationResult(StaticEvaluationResults.Fail,
                string.Format("{0} does not inherit from {1}", type.Name, baseType.Name));
}
```

Figure 28 – Implementing a Static Entry Point

The code shown in Figure 28 makes use of the "GetClassBases" method to get any base types that the class may derive from. After ensuring that there are actually valid types in the bases list, the method iterates through each one and compares the name to the name of the type represented by the second parameter. If the extension finds a match, a new StaticEvaluationResult object is created and the result is set to "StaticEvaluationResults.Pass". If no such match is found, a different StaticEvaluationResult object is created and the result is set to "StaticEvaluationResults.Fail" and a message indicating that the current SUT type does not inherit from the provided SUT type.

Step 10 – Compile and Test

That is all the code you need to create a simple static extension. You should now be able to compile your extension into a .dll file named "InheritFrom.dll". If you receive any sort of error, go back and check to make sure that you completed all of the steps correctly.

Once your can compile your extension into a .dll file, the final step is to have the Contract Evaluation Engine load and recognize your extension. To do this, copy only the "InheritFrom.dll" file from your build folder into the "Extensions" folder which is located wherever you have installed the Contract Editor.

To see if your new extension is recognized by the Contract Evaluation Engine, start the Contract Editor and see if your new extension appears in the "Log Window" as shown in Figure 29.

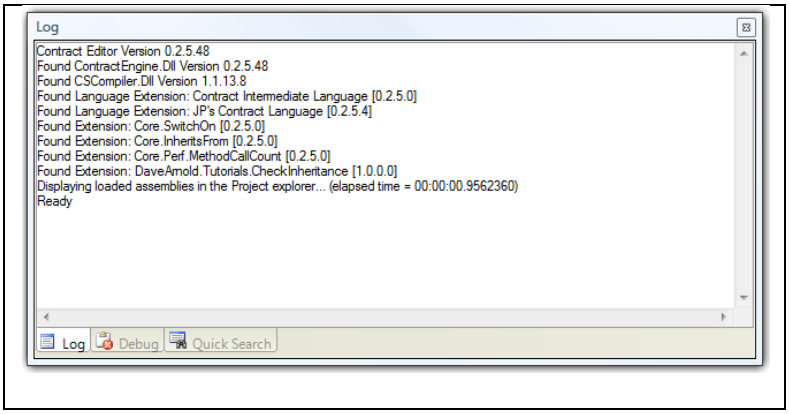


Figure 29 – CheckInheritance Static Extension Found and Loaded

Now that your extension is found and located by the Contract Evaluation Engine, we can use it like any other extension. Using the "Stack.exe" managed executable presented in the Contract Editor Overview Document [2]. We can write the JPCL contract shown in Figure 30.

```
Using DaveArnold.Tutorials;
Contract Stack
{
        Structure
        {
                Belief ~1("All stacks must inherit from IStack");
                CheckInheritance(IStack);
        }
        Exports
        {
                Type IStack;
        }
}
```

Figure 30 – JPCL Contract that uses CheckInheritance

The JPCL code listing creates a contract on the "Stack" type. The "Stack" type will be bound to user supplied type via the binding tool. The contract will define a single belief that states that: "All stacks must inherit from IStack". We will use our new static extension example to verify the belief. The single parameter will be bound to another type in the SUT and referenced as "IStack". For more information on the JPCL syntax please see the JPCL Specification Document [5]. The JPCL compiler will transform the code shown in Figure 30, to the CIL code shown in Figure 31.

```
; CIL Generated By JPCL Compiler 0.2.5.4
; (c) Dave Arnold 2006
; Source File: StackExample.jpcl
; Generated On Tuesday, January 09, 2007 10:37:38 PM
cil             0 2 5 0
; -------------------------- Begin Contract ------------------
con             Stack
bndr            [DaveArnold.Contracts.Examples.Stack]:Stack
bndt            IStack [DaveArnold.Contracts.Examples.Stack]:IStack
str
psh             Stack::1 ; Belief Stack::1
psh             "All stacks must inherit from IStack"
blf
arg             Stack
arg             IStack
call            [DaveArnold.Tutorials]:CheckInheritance
psh             1
eblf
; -------------------------- End Contract --------------------
econ
```

Figure 31 – CIL Contract that uses CheckInheritance

The generated CIL code is then evaluated by the Contract Evaluation Engine, which will actually call our extension and the result will be displayed in a Contract Evaluation Report, similar to the one shown in Figure 32.

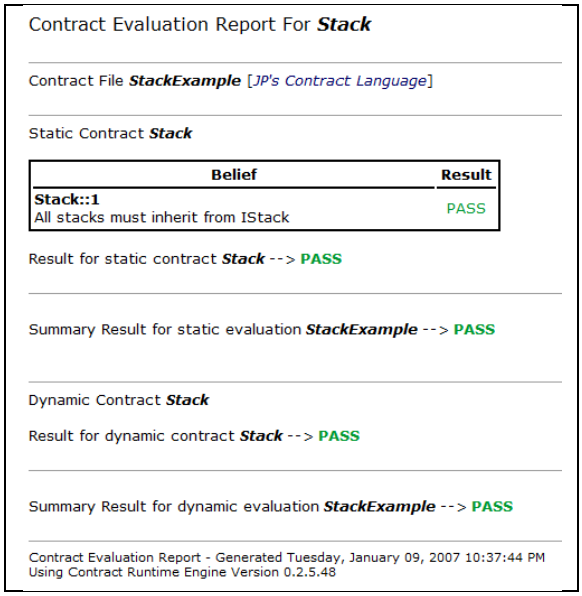


Figure 32 – Contract Evaluation Report for the Contract shown in Figure 31

The execution of a contract which makes use of our CheckInheritance static extension concludes this tutorial. Tutorial number two will focus on creating a static extension that works with a method's behaviour.

Tutorial 2 – SwitchTest

While a single Visual Studio project can be used to contain multiple extensions of various types, for ease of understanding we will create a separate Visual Studio project for each of our tutorials. In order to show that the Contract Evaluation Engine Extension SDK can be used to create extensions using languages other than C#, this tutorial will be implemented using C++.

Step 1 – Create Project

To begin create a new Visual C++ Class Library Project (.dll), and call it SwitchTest, as shown in Figure 33.

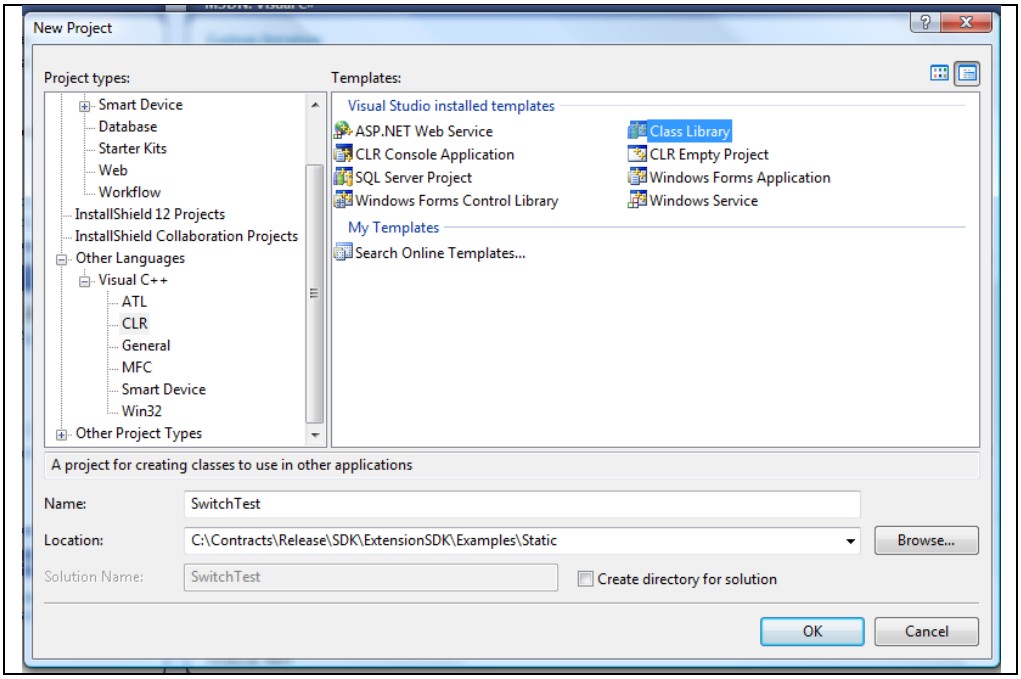


Figure 33 – Creating a new project for SwitchTest

Step 2 – Add References

Once the project has been created, you will be shown an initial code window with a dummy class named Class1. You may delete this class, by selecting both the .h and .cpp file from the Solution Explorer window and pressing the delete key. Before we can begin to create our Static Extension, we need to first reference the Contract Evaluation Engine Extension SDK. To do this, select the "SwitchTest" project in the Solution Explorer window and then right click. Select "References…" from the context menu as shown in Figure 34.

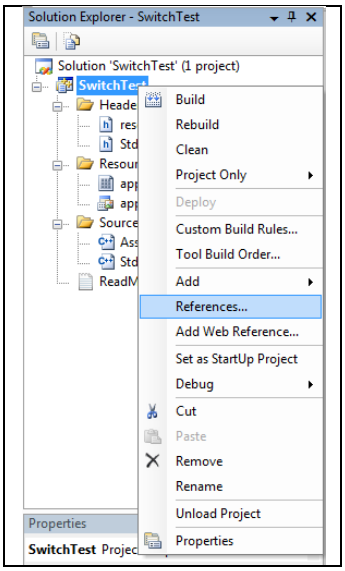


Figure 34 – Adding Contract Evaluation Engine SDK References

When the "SwitchTest Property Pages" dialog box is displayed, select the "Add New Reference…" button, and when the "Add Reference" dialog box is displayed, select the "Browse" tab. In the folder where you have installed the Contract Evaluation Engine Extension SDK, select the "CSCompiler.dll" component file, as shown in Figure 35. Repeat the previous steps again, except this time select the "ExtensionAPI.dll" component file. For the purposes of this example, the CSCompiler.dll file should have a file version of 1.1.13.8, and the ExtensionAPI.dll file should have a file version of 0.2.5.1.

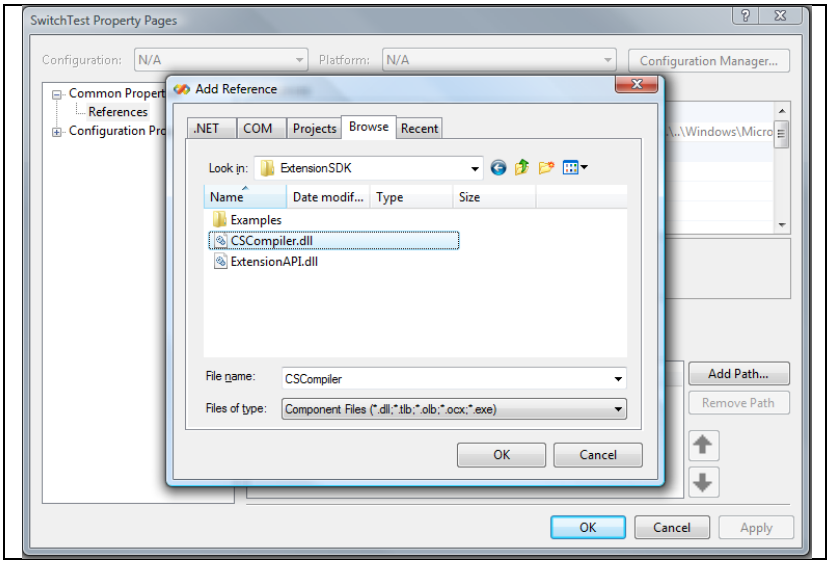


Figure 35 – Selecting SDK Components to Reference

Once the required SDK references have been added, the "SwitchTest Property Pages" dialog should look like the one shown in Figure 36.

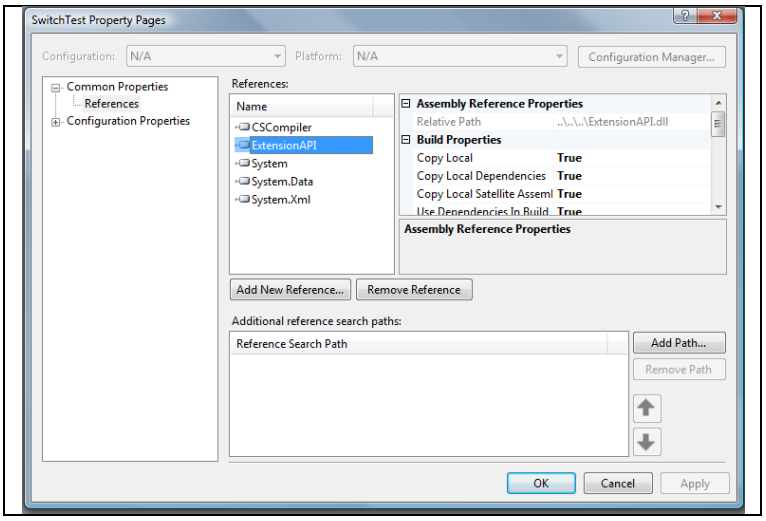


Figure 36 – SDK References Added Correctly

Step 3 – Basic Code Structure

Once the required references have been added to the project, add a new C++ Class to the project, name the class "SwitchTest". To begin the extension, enter the code shown in Figure 37, into the new SwitchTest.h file.

```
#pragma once

using namespace System;
using namespace DaveArnold::Contracts::CSCompiler;
using namespace DaveArnold::Contracts::ExtensionAPI;

namespace DaveArnold
{
	namespace Tutorials
	{
		public ref class SwitchTest : IExtension
		{
			public:

			virtual property String^ Name { String^ get(); }
			virtual property String^ Namespace { String^ get(); }
			virtual property System::Version^
					Version { System::Version^ get(); }
		};
	}
}
```

Figure 37 – Initial SwitchTest.h C++ Code

The first three lines import various namespaces into our code. The "System" namespace contains the core .NET types. The "CSCompiler" namespace contains the types, methods, properties, and attributes for the Class, Field, and Method parameter types already discussed. Finally, the "ExtensionAPI" namespace contains the generic classes required to implement a Contract Evaluation Engine Extension.

Following the using directives, several namespace declarations are used place the new extension into the "DaveArnold::Tutorials" namespace. There is no requirement for the extensions to reside in any particular namespace. Namespaces can be used for organization at the developers discretion, and have no effect on the naming of use of the extension itself. After the namespace, a simple public class is used to contain our extension. Any number of extensions may be present within one class, and the name of the class has no relationship to the name of the extension as you will soon see. It is good practice to name your class the same as the extension name, for code organization purposes, but the Contract Evaluation Engine Extension SDK does not use the class name for extension resolution. Finally, all extensions must reside within a class that is defined public. If the class does not contain the public modifier, the extension will not be recognized or loaded by the Contract Evaluation Engine.

In order for the extension to be recognized by the Contract Engine it must be located in a class that implements the IExtension interface. The IExtension interface is located in the "DaveArnold::Contracts::ExtensionAPI" which we have already imported, so all we need to do is have our class definition looks like this: "public ref class SwitchTest : IExtension".

As shown in the first tutorial, the IExtension interface contains three read-only properties which need to be implemented. Figure 37 illustrates their definition, and Figure 38 shows the corresponding SwitchTest.cpp file.

```
#include "StdAfx.h"
#include "SwitchTest.h"

String^ DaveArnold::Tutorials::SwitchTest::Name::get()
{
        return "SwitchTest";
}
String^ DaveArnold::Tutorials::SwitchTest::Namespace::get()
{
        return "DaveArnold.Tutorials";
}
Version^ DaveArnold::Tutorials::SwitchTest::Version::get()
{
        return gcnew System::Version(1, 0, 0, 0);
}
```

Figure 38 – Implementing the IExtension Interface

The C++ code shown in Figure 38 provides implementations for the three read-only properties defined by the IExtension interface. The implementation for "Name" and "Namespace" is straightforward. The "Version" property's implementation is accomplished by using the "gcnew" keyword to create a new instance of the "System::Version" class. The version number for out new static extension will be: "1.0.0.0".

Step 4 – Define the Static Entry Point

Our extension now has all the required components for it to be a static extension. The problem is that, we have not defined the behaviour for our extension. To do this, we need to create a static entry point. As the name suggests, the entry point will act as our extension's main method. The entry point will be called each time the extension is called from within the Contract Evaluation Engine. To define a static entry point, add the code shown in Figure 39, to the "SwitchTest" class defined in the "SwitchTest.h" file.

```
[StaticEntryPoint()] StaticEvaluationResult^ DoStaticEvaluation(
                                                TypeContainer^ type, Method^ m, Field^ f);
```

Figure 39 – Defining a Static Entry Point

The static entry point shown in Figure 39, takes three parameters. The first parameter is the required context parameter. That is, it represents the current SUT type that the contract is evaluating on. The second parameter will represent a method that is defined in the SUT. For the purposes of our static extension, the method will define the block of code that we will examine for switch statements. Finally, the third parameter will represent a field that is defined within the SUT. This field will be used

when analyzing the method's body to determine if the body contains a switch statement that operates on the given field.

To complete the implementation of the static entry point, add the code shown in Figure 40 to the end of the "SwitchTest.cpp" file.

```
[StaticEntryPoint()] StaticEvaluationResult^
        DaveArnold::Tutorials::SwitchTest::DoStaticEvaluation(
                                                TypeContainer^ type, Method^ m, Field^ f)
{
        // TODO: Implement Static Entry Point
}
```

Figure 40 – Implementation of a Static Entry Point

Before we go ahead and implement the behaviour for our extension, a few things to note. The method name "DoStaticEvaluation" is completely arbitrary. The only required name is the use of the "StaticEntryPoint" attribute. It is the attribute which will alert the Contract Evaluation Engine where the entry point is located. All entry points must have their access modifiers set to public. If a non-public entry point is located, it will be ignored.

As previously discussed, all static entry points must have a return type of "StaticEvaluationResult", and must have at least one parameter of the "TypeContainer" type which will represent the current SUT type we are working with. However, any remaining parameters can be whatever is required in order to perform the desired evaluation as long as they conform to one of the parameter types previously discussed in this document.

The actual SUT types, methods, and fields which will be passed to an extension vary depending on the bindings that the test developer selects during the compilation of the high-level contract language. There is no way to access the binding tables from within any type of extension. Extensions, should function on any type, method, or field, and not assume that they will be passed a specific one each and every time, as binding information can change on every execution.

Step 5 – Implement the Switch On Code Visitor

We would like our static extension to examine the body of the given method and locate any switch statements that may exist in the body, if any. As we have already seen, the statement and expression visitors can be used for just such a purpose.

To create a new specialized visitor, add a new C++ class to the "SwitchTest" project and name the class "SwitchOnCodeVisitor". Once you have created the "SwitchOnCodeVisitor" class, add the code shown in Figure 41 to the "SwitchOnCodeVisitor.h" file.

```
#pragma once
using namespace System;
using namespace DaveArnold::Contracts::CSCompiler;
using namespace DaveArnold::Contracts::ExtensionAPI;

namespace DaveArnold
{
	namespace Tutorials
	{
		ref class SwitchOnCodeVisitor : DeepVisitor
		{
		private:
			Field^ field;
			bool   inSwitch;
			bool   result;
		public:
			SwitchOnCodeVisitor(Method^ m, Field^ f);
			property bool Result { bool get(); }

			virtual void ProcessStatement(Switch^ stmt) override;
			virtual void ProcessExpression(MemberAccess^ exp) override;
			virtual void ProcessExpression(FieldExpr^ exp) override;
		};
	}
}
```

Figure 41 –C++ Code for SwitchOnCodeVisitor.h

The code shown in Figure 41 begins with the inclusion of the .NET core types via the System namespace. The code then includes the CSCompiler, and ExtensionAPI namespaces for extension specific types. A single class is defined which derives from the previously discussed "DeepVisitor" class. Our class defines three private variables as follows: The "field" variable will reference the field that we are looking for. That is, the field reference that the switch statement should operate on. The "inSwitch" Boolean value will be used to keep track if we are currently visiting a switch statement or not. Finally, the "result" Boolean value will contain the result of executing our visitor on the given method's block. The value of executing the visitor can be retrieved via the "Result" read-only accessor also defined in Figure 41.

To complete the implementation of our visitor, we need to provide implementation for the overridden methods defined in the base class. These methods are called when we encounter a switch statement, member access expression, and field expression. To complete the implementation of the specialized visitor add the code shown in Figure 42 to the "SwitchOnVisitor.cpp" file.

```
#include "StdAfx.h"
#include "SwitchOnCodeVisitor.h"

DaveArnold::Tutorials::SwitchOnCodeVisitor::SwitchOnCodeVisitor(Method^ m, Field^ f)
: DeepVisitor(m->Block)
{
	field = f;
	Go();
}
bool DaveArnold::Tutorials::SwitchOnCodeVisitor::Result::get() { return result; }
void DaveArnold::Tutorials::SwitchOnCodeVisitor::ProcessStatement(Switch^ stmt)
{
	inSwitch = true;
	if(stmt->Expr != nullptr)
		stmt->Expr->StaticEvalExpression(this);
	inSwitch = false;
	if(stmt->Sections != nullptr)
	{
		for each(SwitchSection^ ss in stmt->Sections)
		{
			for each(SwitchLabel^ sl in ss->Labels)
				sl->Label->StaticEvalExpression(this);
			ss->Block->StaticEvalStatement(this);
		}
	}
}
void DaveArnold::Tutorials::SwitchOnCodeVisitor::ProcessExpression(MemberAccess^ exp)
{
	DeepVisitor::ProcessExpression(exp);
	try
	{
		This^ t = safe_cast<This^>(exp->Expr);
		if(exp->Identifier == field->ShortName)
			result = true;
	}
	catch(InvalidCastException^ e) { }
}
void DaveArnold::Tutorials::SwitchOnCodeVisitor::ProcessExpression(FieldExpr^ exp)
{
	if(inSwitch)
	{
		if(exp->Name == field->ShortName)
			result = true;
	}
	DeepVisitor::ProcessExpression(exp);
}
```

Figure 42 – Implementation C++ Code for SwitchOnCodeVisitor.cpp

The code in Figure 42, is fairly straightforward, the ProcessStatement method is called when the visitor encounters a "switch" statement in the method body. The method marks our switch flag to indicate that we are in a switch statement. The method then visits the expression that defines the variable that the switch statement operates on. If either a direct field expression or a member access expression is encountered, and the internal switch flag is set. The code checks to see if the field name matches the given field name, and if so, a match has been located and the result flag is set.

Step 6 – Implement the Static Entry Point

To complete the static extension we need to implement the static entry point. To do this, fill in the DoStaticEvaluation method located in the "SwitchTest.cpp" file with the code shown in Figure 43.

```
[StaticEntryPoint()] StaticEvaluationResult^
        DaveArnold::Tutorials::SwitchTest::DoStaticEvaluation(
                                                TypeContainer^ type, Method^ m, Field^ f)
{
        SwitchOnCodeVisitor^ check = gcnew SwitchOnCodeVisitor(m, f);
        if(check->Result)
                return gcnew StaticEvaluationResult(StaticEvaluationResults::Pass);
        else
                return gcnew StaticEvaluationResult(StaticEvaluationResults::Fail,
                        String::Format("No switch statement found in method {0} that
                        operates on variable {1}",
                        m->MethodName->ToString(), f->Name));
}
```

Figure 43 – C++ Code representing the body of the DoStaticEvaluation Entry Point

The code creates and initializes a new instance of our SwitchOnCodeVisitor class. The SwitchOnCodeVisitor's constructor executes the visitor; so that once the initialization has taken place, the "Result" property already has the results of executing the static extension. Depending on the result, the method finishes by returning the corresponding StaticEvaluationResult object. Make sure that you add an "#include" directive to the "SwitchTest.cpp" file to include the "SwitchOnCodeVisitor.h" file.

Step 7 – Compile and Test

That is all the code you need to create a simple static extension. You should now be able to compile your extension into a .dll file named "SwitchTest.dll". If you receive any sort of error, go back and check to make sure that you completed all of the steps correctly.

Once your can compile your extension into a .dll file, the final step is to have the Contract Evaluation Engine load and recognize your extension. To do this, copy only the "SwitchTest.dll" file from your build folder into the "Extensions" folder which is located wherever you have installed the Contract Editor.

To see if your new extension is recognized by the Contract Evaluation Engine, start the Contract Editor and see if your new extension appears in the "Log Window" as shown in Figure 44.

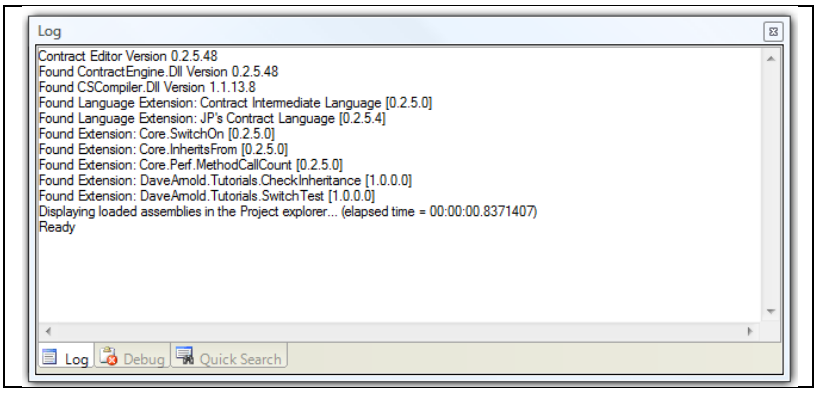


Figure 44 – SwitchTest Static Extension Found and Loaded

Now that your extension is found and located by the Contract Evaluation Engine, we can use it like any other extension. Using the "SwitchTest.exe" managed executable presented in the Contract Editor Overview Document [2]. We can write the JPCL contract shown in Figure 45.

```
Using DaveArnold.Tutorials;
Contract SwitchTestContract
{
        Structure
        {
                Belief ~1("The GoMethod will switch on the 'f' field");
                SwitchTest(GoMethod, f);
        }
        Exports
        {
                Method GoMethod;
                Field f;
        }
}
```

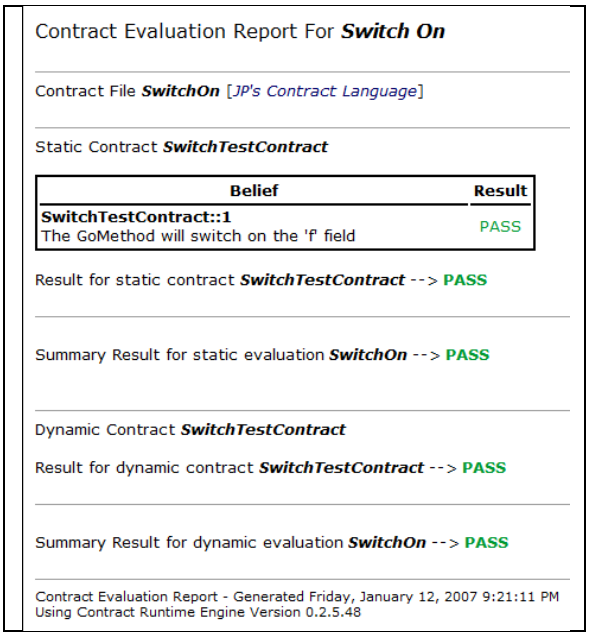
Figure 45 – JPCL Contract that uses SwitchTest

The JPCL code listing creates a contract on the "SwithTestContract" type. The "SwitchTestContract" type will be bound to user supplied type via the binding tool. The contract will define a single belief that states that: "The GoMethod will switch on the 'f' field". We will use our new static extension example to verify the belief. The first parameter will be bound to a method defined in the SUT. The second parameter will be bound to a field defined in the SUT. For more information on the JPCL syntax please see the JPCL Specification Document [5]. The JPCL compiler will transform the code shown in Figure 45, to the CIL code shown in Figure 46.

```
; CIL Generated By JPCL Compiler 0.2.5.4
; (c) Dave Arnold 2006
; Source File: SwitchOn.jpcl
; Generated On Friday, January 12, 2007 9:21:09 PM
cil             0 2 5 0
; -------------------------- Begin Contract ------------------
con             SwitchTestContract
bndr            [DaveArnold.Contracts.Examples.SwitchOn]:SwitchOn
bndm            GoMethod
                [DaveArnold.Contracts.Examples.SwitchOn.SwitchOn]:
                int32 Go()
bndf            f
                [DaveArnold.Contracts.Examples.SwitchOn.SwitchOn]:
                int32 switchVar
str
psh             SwitchTestContract::1 ; Belief SwitchTestContract::1
psh             "The GoMethod will switch on the 'f' field"
blf
arg             SwitchTestContract
arg             GoMethod
arg             f
call            [DaveArnold.Tutorials]:SwitchTest
psh             1
eblf
; -------------------------- End Contract --------------------
econ
```

Figure 46 – CIL Contract that uses SwitchTest

The generated CIL code is then evaluated by the Contract Evaluation Engine, which will actually call our extension and the result will be displayed in a Contract Evaluation Report, similar to the one shown in Figure 47.



Contract Evaluation Report For ***Switch On***

Contract File ***SwitchOn*** [*JP's Contract Language*]

Static Contract ***SwitchTestContract***

Belief	Result
SwitchTestContract::1 The GoMethod will switch on the 'f' field	PASS

Result for static contract ***SwitchTestContract*** --> **PASS**

Summary Result for static evaluation ***SwitchOn*** --> **PASS**

Dynamic Contract ***SwitchTestContract***

Result for dynamic contract ***SwitchTestContract*** --> **PASS**

Summary Result for dynamic evaluation ***SwitchOn*** --> **PASS**

Contract Evaluation Report - Generated Friday, January 12, 2007 9:21:11 PM
Using Contract Runtime Engine Version 0.2.5.48

Figure 47 – Contract Evaluation Report for the Contract shown in Figure 46

The execution of a contract which makes use of our SwitchTest static extension concludes this tutorial. The Contract Evaluation Engine Extension SDK also contains a C# version of this tutorial which can be found in the SwitchTestCSharp folder.

Performance Extensions

Performance extensions are called by code placed in the metrics section of a contract. These extensions contain a fixed number and type of parameters. A performance extension uses the given metric information to compute information that will be displayed on the Contract Evaluation Report. As with static extensions, a class which implements a performance extension, must implement the IExtension interface and must define one or more dynamic entry points. Dynamic entry points are marked methods which are called by the Contract Evaluation Engine, with metric data parameters. The dynamic entry point is responsible for executing the extension's behaviour and writing any findings to the Contract Evaluation Report. Access to the Contract Evaluation Report is provided through the first parameter to the extension, this parameter has the type IMetricReporter which will be discussed shortly. The Contract Evaluation Engine Extension SDK provides an attribute class named DynamicEntryPoint. The attribute can be applied to methods to indicate that the given method will be a dynamic entry point. Figure 48, illustrates a signature for such an entry point.

```
[DynamicEntryPoint(MetricTypes.Performance)]
public void DoEvaluaton(IMetricReporter reporter, IProcessInfoCollection processes)
{
        // To Do: Put code to implement the static extension here
}
```

Figure 48 – Dynamic Entry Point Signature

The entry point shown in Figure 48 begins with the application of the DynamicEntryPoint attribute to the method. The attribute informs the Contract Evaluation Engine that this method can be used as an entry to the performance extension defined by the IExtension.Name property. Unlike static entry points, the dynamic entry point attribute requires a single parameter. The parameter is used to indicate what type of dynamic extension this entry point represents. In the current version of the Contract Evaluation Engine Extension SDK only performance extensions are supported. As such the only acceptable parameter to the DynamicEntryPoint attribute is MetricTypes.Performance. Other dynamic entry point types will be added in future versions of the Contract Evaluation Engine Extension SDK. If the MetricTypes.Unknown parameter is used the dynamic entry point will be ignored by the Contract Engine.

All methods marked with the DynamicEntryPoint attribute must have a specific signature. The signature is based on the type of dynamic entry point being created. For the purposes of this document we will only focus on the performance type of extension. All performance extension entry points must define a method similar to the one shown in Figure 48. That is, the method returns void, and contains two parameters. The first parameter is of type IMetricReporter which provides access to the Contract Evaluation Report. The second parameter is of type IProcessInfoCollection and is used to access the metric information gathered while profiling the SUT. Both parameter types will be discussed in detail in the following subsections.

IMetricReporter

The IMetricReporter interface provides the performance extension with the ability to add documentation to the Contract Evaluation Report. Any information which is added by the performance extension will be placed in the "Dynamic Contract" section. IMetricReporter is located in the DaveArnold.Contracts.ExtensionAPI namespace. The interface definition is shown in Figure 49.

```
namespace DaveArnold.Contracts.ExtensionAPI
{
        public interface IMetricReporter
        {
                void ReportMetric(string html);
        }
}
```

Figure 49 – IMetricReporter Interface

As shown in Figure 49, the IMetricReporter interface only defines a single method for writing content to the Contract Evaluation Report. The method is described by the following table.

ReportMetric

Name:	**ReportMetric**
Description:	**Writes information to the Contract Evaluation Report**
Parameters:	1
Parameter Values:	<table><tr><th>Ordinal</th><th>Type</th><th>Description</th></tr><tr><td>1</td><td>String</td><td>Html text to add to the report</td></tr></table>
Remarks:	The *ReportMetric* method adds the specified html text to the Contract Evaluation Report. The text should contain valid html, and no error checking is done to ensure that the html being added to the Contract Evaluation Report is valid. There is no limit to the amount of text that you may add to the Contract Evaluation Report. The text to be added will be added under the dynamic contract section in the evaluation report. As all performance extensions are independent there is no method for accessing information already in the Contract Evaluation Report.
C# Example:	See code below

```
private void ProcessSingleFunction(IMetricReporter report,
                                   IFunctionInfo fi)
{
        string callList = "";
        foreach (ICalleeFunctionInfo c in fi.CalleeInfo)
        {
                if (c.Calls / fi.Calls == 1)
                {
                        callList += "<li>" + c.Signature + " - " +
                                        (c.Calls / fi.Calls) + " call";
                }
                else
                        callList += "<li>" + c.Signature + " - " +
                                (c.Calls / fi.Calls) + " calls";
                callList += string.Format(" (<i>{0}% of total</i>)",
                c.PercentOfParentTimeInMethod.ToString(
                        "0.00;-0.00;0"));
                callList += "</li>";
        }
        report.ReportMetric("<Font size=-1><B>" +
                fi.Signature.Signature + "</b>:</font><br>" +
                string.Format("<ul><li>{0} calls</li><li>Child
                Calls:</li><ol>{1}</ol><li>{2}% of total</li></ul>",
                fi.Calls, callList,
                fi.PercentOfTotalTimeInMethodAndChildren.ToString(
                "0.00;-0.00;0")));
}
```

Example Notes:	The example method uses the *ReportMetric* method to add method call information to the Contract Evaluation Report. The informationt that the example adds will look as follows: Dynamic Contract ***SwitchOnTest*** **int32 DaveArnold.Contracts.Examples.SwitchOn.SwitchOn::Go():** • 1 calls • Child Calls: • 0.01% of total Result for dynamic contract ***SwitchOnTest*** --> **PASS** A complete example using the method shown above can be found in Tutorial 3.
Thread Safe:	Yes
C# Method Signature:	void ReportMetric(string html);
Version Information:	New in Version 0.2.5.1 of the SDK

IProcessInfoCollection

The IProcessInfoCollection parameter is the root of a whole tree of interfaces which are used to represent the data recorded by the profiler while the SUT was being profiled. By traversing through the tree structure a performance extension is able to process and interpret the metric data then write the result to the Contract Evaluation Report via the IMetricReporter interface.

All metric data interfaces are located in the DaveArnold.Contracts.ExtensionAPI.RuntimeData namespace and can be accessed by traversing the tree rooted at the IProcessInfoCollection type, which is the second parameter passed to all performance extensions. The IProcessInfoCollection interface is shown in Figure 50.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
        public interface IProcessInfoCollection : IEnumerable
        {
                void Add(object o);
                IProcessInfo this[int nProcessID] { get; }
        }
}
```

Figure 50 – IProcessInfoCollection Interface

As Figure 50 shows, the IProcessInfoCollection interface is simply a container for the IProcessInfo objects. The objects can be iterated through via a "foreach" loop or you can access individual process information via the interface's indexer using the process identifier. The Add method can be used to add additional IProcessInfo objects to the collection. The Add method does not really have a use when writing performance extensions so it will not be discussed any further in this document. The next section will examine the IProcessInfo type in more detail.

IProcessInfo

The IProcessInfo interface represents a single process used to execute the SUT. The IProcessInfo interface is used as a container to hold the threads which make up the process, as well as other process identifying information. Figure 51 shows the IProcessInfo interface.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
        public interface IProcessInfo
        {
                IFunctionSignatureMap Functions { get; }
                string Name { get; }
                int ProcessID { get;}
                IThreadInfoCollection Threads { get; }
        }
}
```

Figure 51 – IProcessInfo

A process information object contains four read-only properties. The "Name" and "ProcessID" properties provide the process' textual name and process identifier. The "Threads" property is used to get the collection of threads that compose the process. Finally, the "Functions" property provides access to a IFunctionSignatureMap object. Function signature maps are used to map a method's unique identifier to a textual representation of the method for display purposes.

IFunctionSignatureMap

The IFunctionSignatureMap interface is used to provide a map between a method identifier and a textual representation of the method. A method identifier is simply a unique integer used to reference a given method. Figure 52 shows the listing of the IFunctionSignatureMap interface.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
        public interface IFunctionSignatureMap
        {
                string GetFunctionSignature(int nFunctionID);
                void MapSignature(int nFunctionID, IFunctionSignature fs);
        }
}
```

Figure 52 – IFunctionSignatureMap

The IFunctionSignatureMap interface contains two methods. The first method shown in Figure 52, GetFunctionSignature, is used to convert a method identifier into a string representation of the method. This method is useful when writing method names to the Contract Evaluation Report. The second method which is seldom used in a performance extension itself is used to add a new method signature to the map and is used by the SUT profiler. Method signatures are specified via the IFunctionSignature interface.

IFunctionSignature

The IFunctionSignature interface represents a read-only container for a single method. The function's attributes such as return type and parameter set can be retrieved by accessing the various properties defined by the interface. A listing of the interface is shown in Figure 53.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
        public interface IFunctionSignature
        {
                string ClassName { get; }
                string FunctionName { get; }
                bool IsExtern { get; }
                bool IsPInvoke { get; }
                bool IsStatic { get; }
                string[] Namespace { get; }
                string NamespaceString { get; }
                string Parameters { get; }
                string ReturnType { get; }
                string Signature { get; }
        }
}
```

Figure 53 – IFunctionSignature

The "ClassName" property will return a string representation of the class name that contains the method. The "FunctionName" property will return the name of the method without the return type or parameter list. The "IsExtern" property will return true if the method is defined in an external .dll file and the method is simply a definition entry. The "IsPInvoke" property will return true if the method is a COM method and will be called via using the P/Invoke method, defined under COM. The "IsStatic" property will return true if the method is defined using the static modifier. The "Namespace" property will return an array of each section of the namespace name that contains the method. That is, if the method is located in the "DaveArnold.Examples.Stack" namespace, then the "Namespace" property will return a list containing the following elements: "DaveArnold", "Examples", and "Stack". The "NamespaceString" property returns a single string representation of the method's namespace. The "Parameters" property returns a string that contains a list of the parameters which the method accepts. The "ReturnType" property returns the name of the type that the method returns. Finally, the "Signature" property returns a single string that contains all of the information found in the other properties.

IThreadInfoCollection

The IThreadInfoCollection interface is used as a container for IThreadInfo objects. As shown in Figure 54, the IThreadInfoCollection interface is IEnumerable so that developers can use the "foreach" statement to iterate through each of the IThreadInfo objects that make up the collection. IThreadInfo objects can also be accessed directly via the IThreadInfoCollection's indexer using the thread identifier. The IThreadInfoCollection interface also contains an "Add" method which is used to add new

IThreadInfo objects to the collection. The "Add" method is used by the Contract Engine to populate the collection with the IThreadInfo objects prior to calling any performance extensions.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
        public interface IThreadInfoCollection : IEnumerable
        {
                void Add(object o);
                IThreadInfo this[int nThreadID] { get; }
        }
}
```

Figure 54 – IThreadInfoCollection

IThreadInfo

The IThreadInfo interface represents a single thread. The interface only contains information relevant to performance metrics. That is, only information related to running times is included in the interface. As additional types of dynamic extensions are supported via the Contract Evaluation Engine Extension SDK, the IThreadInfo interface will contain additional methods and properties. Figure 55 lists the current version of the IThreadInfo interface.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
        public interface IThreadInfo
        {
                long EndTime { get; }
                IFunctionInfoCollection FunctionInfoCollection { get; }
                long StartTime { get; }
                long TotalTime { get; }
        }
}
```

Figure 55 – IThreadInfo

The "StartTime", "EndTime", and "TotalTime" properties report time intervals. The "StartTime" property contains the time when the thread was started. Likewise, the "EndTime" property contains the time when the thread finished. The "TotalTime" property, indicates the total amount of time that the system spent executing the thread.

The "FunctionInfoCollection" property returns a collection of FunctionInfo objects which represent the methods that were called by the thread. The next sections will look at the IFunctionInfoCollection and IFunctionInfo interfaces respectively.

IFunctionInfoCollection

The IFunctionInfoCollection interface is shown in Figure 56, and represents a collection of IFunctionInfo objects. The IFunctionInfoCollection interface is IEnumerable and thus can be iterated through via the "foreach" keyword. The interface only contains two "Add" methods which are used by the Contract Engine to populate the collection as the SUT is being profiled.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
	public interface IFunctionInfoCollection : IEnumerable
	{
		void Add(object o);
		void Add(IFunctionInfo fi);
	}
}
```

Figure 56 – IFunctionInfoCollection

IFunctionInfo

The IFunctionInfo interface represents performance metric information for a single method. The listing of the IFunctionInfo interface is shown in Figure 57. The interface contains several properties used to access performance metric information about the method.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
	public interface IFunctionInfo
	{
		ICalleeFunctionInfo[] CalleeInfo { get; }
		int Calls { get; }
		double PercentOfMethodTimeInChildren { get; }
		double PercentOfMethodTimeSuspended { get; }
		double PercentOfTotalTimeInChildren { get; }
		double PercentOfTotalTimeInMethod { get; }
		double PercentOfTotalTimeInMethodAndChildren { get; }
		double PercentOfTotalTimeSuspended { get; }
		IFunctionSignature Signature { get; }
		long ThreadTotalTicks { get; }
		long TotalChildrenRecursiveTicks { get; }
		long TotalChildrenTicks { get; }
		long TotalRecursiveTicks { get; }
		long TotalSuspendedTicks { get; }
		long TotalTicks { get; }
		bool IsMarked { get; }
	}
}
```

Figure 57 – IFunctionInfo

The "CalleeInfo" property returns an array of ICalleeFunctionInfo objects. Each one of these objects represents a method which is called by the current method represented by the IFunctionInfo object. The execution time for the current method includes the time spent executing the methods returned by the "CalleeInfo" property.

The "Calls" property returns the number of times that the current method has been called during the thread's lifetime. Each IFunctionInfo object only appears in the collection once no matter how many times the method represented by the IFunctionInfo object is called.

The "PercentOfMethodTimeInChildren" property returns a percentage value indicating the amount of time the method spends executing child methods, rather than the method body itself.

The "PercentOfMethodTimeSuspended" property returns a percentage value indicating the amount of time the method spends waiting in a suspended state. Methods will be in a suspended state while waiting for I/O operations to complete, or while a garbage collection operation is taking place. As the SUT will be executed within a managed environment, the value returned by the "PercentOfMethodTimeSuspended" property may change over different executions.

The "PercentOfTotalTimeInChildren" property returns a percentage value indicating the total amount of application time that the method spends executing child methods, which are called from the original method body itself. Total time is different from method time in that, method time only includes time spend executing the thread that is running the method, where total time is total application time.

The "PercentOfTotalTimeInMethod" property returns a percentage value indicating the total amount of application time spent executing the given method. This time value only includes the time spend executing the method itself, and not any child methods.

The "PercentOfTotalTimeInMethodAndChildren" property returns a percentage value indicating the total amount of application time spent executing the method. This time value includes both the time spent executing the method itself as well as any children that the method may have.

The "PrecentOfTotalTimeSuspended" property returns a percentage value indicating the amount of application time that the method spends in the suspended state.

The "Signature" property returns an object of the previously discussed IFunctionSignature type. This object contains signature information for the method itself. It is useful when reporting the method signature to the Contract Evaluation Report.

The "ThreadTotalTicks" property returns the number of ticks that have elapsed while the thread used to execute this method has been running. This number includes the total number of ticks, not just the ticks used to execute the method.

The "TotalChildrenRecursiveTicks" property returns the number of ticks spent executing all of the child methods contained within the current method. The property is recursive and the children's children, etc. will also be included in the tick count.

The "TotalChildrenTicks" property returns the number of ticks spent executing all of the child methods contained within the current method. The property will not include any ticks devoted to recursive child calls.

The "TotalRecursiveTicks" property returns the number of ticks spent executing the method and any recursive calls needed to complete the initial method execution.

The "TotalSuspendedTicks" property returns the number of ticks that the thread spent while in a suspended state during the execution of the method. Methods will be in a suspended state while waiting for I/O operations to complete, or while a garbage collection operation is taking place. As the SUT will be executed within a managed environment, the value returned by the "TotalSuspendedTicks" property may change over different executions.

The "TotalTicks" property returns the total number of ticks that have elapsed during the execution of this method. The "TotalTicks" property includes the number of ticks spent executing any child methods as well as any ticks spent in a suspended state.

The "IsMarked" property is used to indicate that the Contract Engine requires metric analysis and reporting for this method. The property will return a true value when the user specifies a contract that requires performance metric analysis of this method. Developers should take the "IsMarked" property into account while writing performance extensions. If one or more methods are marked, then only those marked methods should be analyzed and reported on. If no such methods are marked, then all methods contained within the IFunctionInfoCollection object should be analyzed and reported on.

ICalleeFunctionInfo

The ICalleeFunctionInfo interface represents a child method call. The interface listing for the ICalleeFunctionInfo type is shown in Figure 58.

```
namespace DaveArnold.Contracts.ExtensionAPI.RuntimeData
{
        public interface ICalleeFunctionInfo
        {
                int Calls { get; }
                double PercentOfParentTimeInMethod { get; }
                double PercentOfTotalTimeInMethod { get; }
                string Signature { get; }
                long TotalRecursiveTime { get; }
                long TotalTime { get; }
        }
}
```

Figure 58 – ICalleeFunctionInfo

The ICalleeFunctionInfo provides several read-only properties to gain additional about the child method call. The "Calls" property indicates the number of times that the child method has been called by the parent. This value is the number of times that the parent method calls the child per execution.

The "PercentOfParentTimeInMethod" property returns a percentage value which indicates the amount of time that the parent spends executing the child method.

The "PercentOfTotalTimeInMethod" property returns a percentage value which indicates the total amount of application time that the method uses.

The "Signature" property returns a string representation of the method signature. The string is useful when reporting the method signature to the Contract Evaluation Report.

The "TotalRecursiveTime" property returns the amount of time in ticks that the method spends executing recursive calls to itself.

The "TotalTime" property returns the total amount of application time in ticks that the method spends executing, being suspended, or making child calls.

Using Performance Extensions

Performance extensions do not operate the same way as static extensions. That is, performance extensions are not explicitly called from within a contract. This section will discuss how to load and call a performance extension. If a performance section is marked with the correct DynamicEntryPoint attribute, but does not contain the pre-described signature, an extension load error will be reported in the Contract Editor's Log window, and the entire extension will be unusable.

Performance extensions are triggered via the use of the JPCL "Performance" keyword [5]. The "Performance" keyword instructs the Contract Engine that the method pattern specified by the parameter is to be profiled by the SUT and metric information should be analyzed and reported. Any referenced performance extensions will be automatically called by the Contract Engine once profiling of the SUT has been completed. A performance extension is referenced by including the namespace where the performance extension resides via the JPCL "Using" keyword. An example, of a contract which uses the "Performance" keyword is shown in Figure 59.

```
Using Core.Perf;
Contract SwitchOnTest
{
        Metrics
        {
                Performance(dontcare Go(void));
        }
}
```

Figure 59 – JPCL listing for calling a performance extension

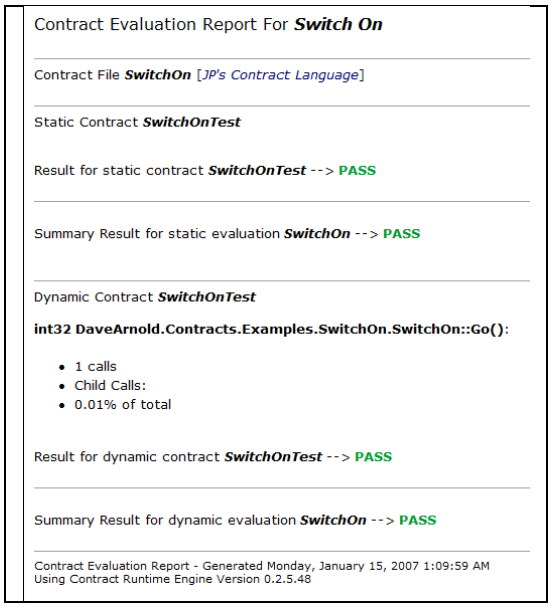
The JPCL contract will generate performance metrics for any method named "Go" defined within the contract's containing type that takes zero parameters. The return type does not matter because of the use of the "dontcare" keyword. The Contract Engine will then automatically execute any performance extensions defined within the "Core.Perf" namespace to process and analyze and report on the metric information generated by the profiler. The JPCL compiler will generate the CIL code shown in Figure 60 [1].

```
; CIL Generated By JPCL Compiler 0.2.5.4
; (c) Dave Arnold 2006
; Source File: SwitchOn.jpcl
; Generated On Monday, January 15, 2007 1:09:57 AM
cil             0 2 5 0
; -------------------------- Begin Contract ------------------
con             SwitchOnTest
bndr            [DaveArnold.Contracts.Examples.SwitchOn]:SwitchOn
met
pshm            ?:Go:null
cnt
; -------------------------- End Contract --------------------
econ
```

Figure 60 – CIL listing for the contract shown in Figure 59

The Contract Engine will then execute the CIL listing and generate the Contract Evaluation Report shown in Figure 61.

***NOTE**: The CIL listing shown in Figure 60 does not contain a reference to the "Core.Perf" namespace or a specific call to the performance extension. While JPCL contracts work because of the compiler/method call tables which remain intact during CIL execution, a purely CIL contract would not execute the performance extension. This is a bug and will be corrected in the next build of the Contract Editor. JPCL contracts are not affected and can be used correctly; this only applies to CIL only contracts.*



Contract Evaluation Report For ***Switch On***

Contract File ***SwitchOn*** [*JP's Contract Language*]

Static Contract ***SwitchOnTest***

Result for static contract ***SwitchOnTest*** --> **PASS**

Summary Result for static evaluation ***SwitchOn*** --> **PASS**

Dynamic Contract ***SwitchOnTest***

int32 DaveArnold.Contracts.Examples.SwitchOn.SwitchOn::Go():

- 1 calls
- Child Calls:
- 0.01% of total

Result for dynamic contract ***SwitchOnTest*** --> **PASS**

Summary Result for dynamic evaluation ***SwitchOn*** --> **PASS**

Contract Evaluation Report - Generated Monday, January 15, 2007 1:09:59 AM
Using Contract Runtime Engine Version 0.2.5.48

Figure 61 – Contract Evaluation Report for the contract shown in Figures 59 & 60

The document will conclude with a full tutorial for creating a performance extension.

Performance Extension Tutorials

The third tutorial will consist of creating a simple performance extension to report method call performance metrics directly to the Contract Evaluation Report.

Tutorial 3 – MethodCallTest

While a single Visual Studio project can be used to contain multiple extensions of various types, for ease of understanding we will create a separate Visual Studio project for each of our tutorials.

Step 1 – Create Project

To begin create a new Visual C# Class Library Project (.dll), and call it MethodCallTest, as shown in Figure 62.

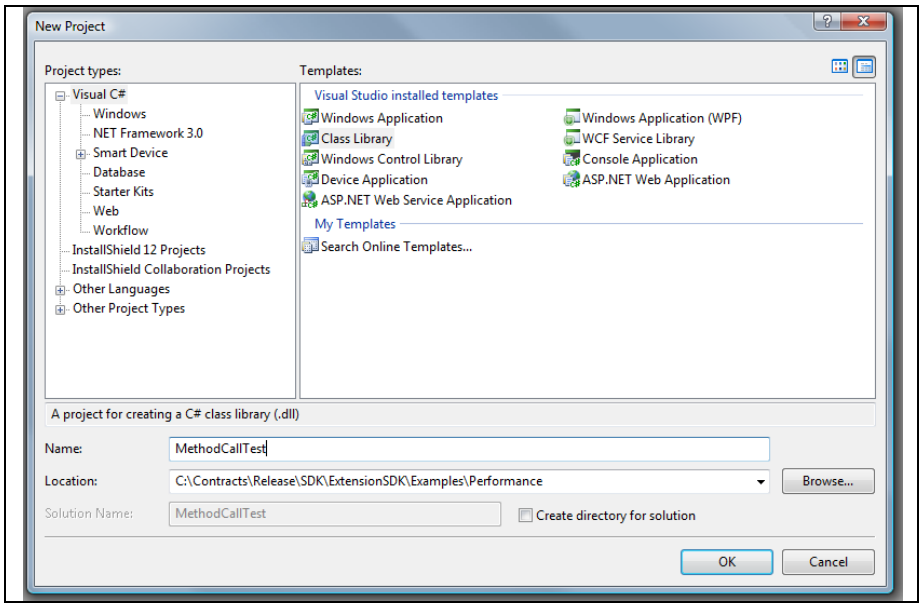


Figure 62 – Creating a new project for MethodCallTest

Step 2 – Add References

Once the project has been created, you will be shown an initial code window with a dummy class named Class1.cs. You may delete this class, by selecting it from the Solution Explorer window and pressing the delete key. Before we can begin to create our Performance Extension, we need to first reference the Contract Evaluation Engine Extension SDK. To do this, select the "References" folder in the Solution Explorer window and then right click. Select "Add Reference…" from the context menu as shown in Figure 63.

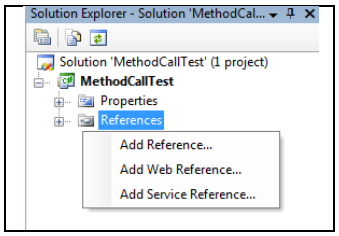


Figure 63 – Adding Contract Evaluation Engine SDK References

When the "Add Reference" dialog box is displayed, select the "Browse" tab, and in the folder where you have installed the Contract Evaluation Engine Extension SDK, select the "ExtensionAPI.dll" component file. When the component is selected, as shown in Figure 64, click the "OK" button to add the required reference. For the purposes of this example, the ExtensionAPI.dll file should have a file version of 0.2.5.1.

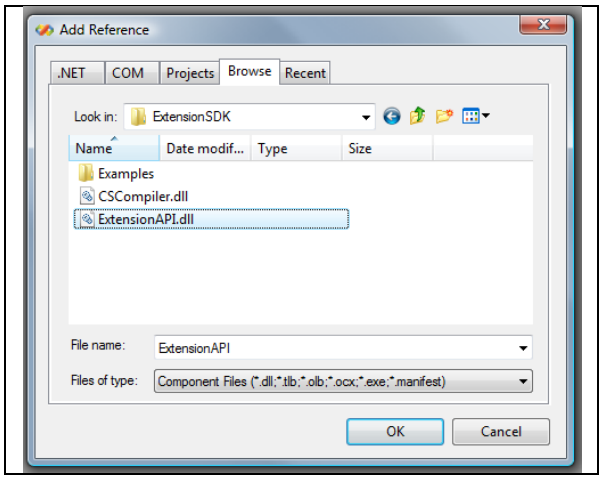


Figure 64 – Selecting SDK Components to Reference

Once the required SDK references has been added, the "References" folder located in the Solution Explorer should look like the one shown in Figure 68.

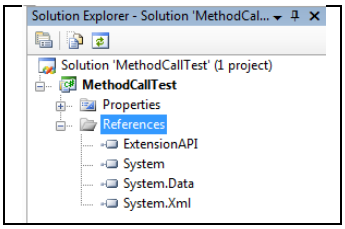


Figure 65 – SDK Reference Added Correctly

Step 3 – Basic Code Structure

Once the required reference has been added to the project, add a new C# code file to the project, name the file "MethodCallTest.cs". To begin the extension, enter the code shown in Figure 66, into your new code file.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using DaveArnold.Contracts.ExtensionAPI;
using DaveArnold.Contracts.ExtensionAPI.RuntimeData;

namespace DaveArnold.Contracts.Examples.MethodCallTest
{
        public class MethodCallCount
        {
        }
}
```

Figure 66 – Initial MethodCallCount C# Code

The first six lines import various namespaces into our code. The "System" namespace contains the core .NET types. The "System.Collections.Generic" namespace includes support for various templated collection types. The "System.Text" namespace contains text handling and manipulation types. The "System.Reflection" namespace contains types for examining the internal makeup of an assembly. Finally, the "ExtensionAPI", and "ExtensionAPI.RuntimeData" namespaces contain all the generic classes required to implement a Contract Evaluation Engine Extension and access the runtime data generated by the profiler.

Following the using directives, a namespace is used to group the extensions together. There is no requirement for the extensions to reside in any particular namespace. Namespaces can be used for organization at the developers discretion, and have no effect on the naming of use of the extension itself. After the namespace, a simple public class is used to contain our extension. Any number of extensions may be present within one class, and the name of the class has no relationship to the name of the extension as you will soon see. It is good practice to name your class the same as the extension name, for code organization purposes, but the Contract Evaluation Engine Extension SDK does not use the class name for extension resolution. Finally, all extensions must reside within a class that is defined public. If the class does not contain the public modifier, the extension will not be recognized or loaded by the Contract Evaluation Engine.

Step 4 – Implement IExtension

In order for the extension to be recognized by the Contract Engine it must be located in a class that implements the IExtension interface. The IExtension interface is located in the "DaveArnold.Contracts.ExtensionAPI" which we have already imported, so all we need to do is change our class definition so it looks like this: "public class MethodCallCount : IExtension". Make this change now.

When the IExtension interface is added to the class declaration, Visual Studio will mark the first letter of the interface name (in our case the "I"), with a blue underline. If you do not see a blue underline, simply click on the "I" in the "IExtension" interface name. If you click on the blue underline, you will get the menu shown in Figure 67.

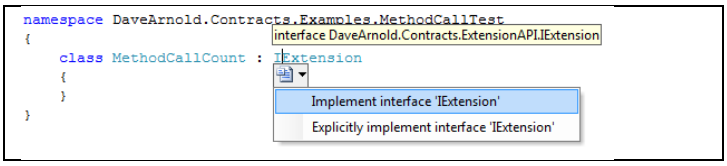


Figure 67 – Implementing the IExtension Interface

From the context menu that appears select the "Implement interface 'IExtension'" menu item. Visual Studio will now automatically insert members that will implement the IExtension interface into your new MethodCallCount class. If everything was done correctly, your MethodCallCount.cs should look like the one shown in Figure 68.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using DaveArnold.Contracts.ExtensionAPI;
using DaveArnold.Contracts.ExtensionAPI.RuntimeData;

namespace DaveArnold.Contracts.Examples.MethodCallTest
{
        class MethodCallCount : IExtension
        {
                #region IExtension Members

                public string Name
                {
                        get { throw new Exception("The method or operation is not
                                implemented."); }
                }

                public string Namespace
                {
                        get { throw new Exception("The method or operation is not
                                implemented."); }
                }

                public Version Version
                {
                        get { throw new Exception("The method or operation is not
                                implemented."); }
                }

                #endregion
        }
}
```

Figure 68 –MethodCallTest C# Code after completing Step 4

Step 5 – Implement IExtension.Name

As shown in Figure 68, the IExtension interface only contains three read only properties that need to be implemented. The first is the Name property. As the name suggests, the Name property must return the name of the extension. This is the name that will be used to identify the extension. As such, the name returned by the "Name" property must conform to a valid C++ method call. That is, the name cannot contain any spaces or punctuation symbols, and cannot begin with a number. Extension names may begin with underscores if desired. If an invalid extension name is provided, the extension will be ignored by the Contract Evaluation Engine.

To name our extension "MethodCallTest" replace the body of the Name property's get accessor with the code shown in Figure 69.

```
public string Name
{
        get { return "MethodCallTest"; }
}
```

Figure 69 – Naming the extension MethodCallTest

Step 6 – Implement IExtension.Namespace

The Namespace property is used to indicate where the extension resides. Extensions are organized into namespaces for ease of use and grouping. Extension namespaces have the same rules are C# namespaces. Each namespace cannot contain any spaces or punctuation characters. Namespaces are separated via the dot "." character.

In order to keep our simple tutorial extensions separate from the production ones, we will place our new extension in the "DaveArnold.Tutorials" namespace. To do this, replace the body of the Namespace property's get accessor with the code shown in Figure 70.

```
public string Namespace
{
        get { return "DaveArnold.Tutorials.Perf"; }
}
```

Figure 70 – Setting the Extension's Namespace to DaveArnold.Tutorials.Perf

Step 7 – Implement IExtension.Version

The final property to implement is the Version property. The Version property specifies the version number of the extension. Versions are represented by the System.Version structure, and consist of four numbers: major, minor, build, and revision numbers. The Version structure contains several constructors which allow you to set any number of the version components.

Unlike the previous tutorials, we will not statically specify a specific version. Instead we will use the assembly version number contained within our MethodCallTest.Dll file to identify the version of our performance extension. To accomplish this, replace the body of the Version property's get accessor with the code shown in Figure 71.

```
public Version Version
{
        get
        {
                object[] attribs =
                System.Reflection.Assembly.GetAssembly(
                        this.GetType()).GetCustomAttributes(true);
                for (int i = 0; i < attribs.Length; i++)
                {
                        if (attribs[i] is AssemblyFileVersionAttribute)
                        {
                                AssemblyFileVersionAttribute av = attribs[i] as
                                        AssemblyFileVersionAttribute;
                                return new Version(av.Version);
                        }
                }
                return new Version(-1, -1, -1, -1);
        }
}
```

Figure 71 – Setting the Extension's Version to the Assembly Version

Step 8 – Define the Performance Entry Point

Our extension now has all the required components for it to be a performance extension. The problem is that, we have not defined the behaviour for our extension. To do this, we need to create a performance entry point. As the name suggests, the entry point will act as our extension's main method. The entry point will be called each time the extension is called from within the Contract Evaluation Engine. To define a performance entry point, add the code shown in Figure 72, to your MethodCallTest.cs code file.

```
[DynamicEntryPoint(MetricTypes.Performance)]
public void DoMethodCallCount(IMetricReporter reporter, IProcessInfoCollection processes)
{
}
```

Figure 72 – Defining a Performance Entry Point

The performance entry point shown in Figure 72, takes two parameters matching the required method signature for performance extensions. Before we go ahead and implement the behaviour for our extension, a few things to note. The method name "DoMethodCallCount" is completely arbitrary. The only required name is the use of the "DynamicEntryPoint" attribute. It is the attribute which will alert the Contract Evaluation Engine where the entry point is located. All entry points must have their access modifiers set to public. If a non-public entry point is located, it will be ignored.

Step 9 – Implement the Performance Entry Point

To complete the performance extension we need to implement the entry point. To do this, fill in the DoMethodCallCount method located in the MethodCallCount.cs file with the code shown in Figure 73.

```
[DynamicEntryPoint(MetricTypes.Performance)]
public void DoMethodCallCount(IMetricReporter reporter, IProcessInfoCollection processes)
{
        bool oneMarked = false;
        foreach (IProcessInfo pi in processes)
        {
                foreach (IThreadInfo ti in pi.Threads)
                {
                        foreach (IFunctionInfo fi in ti.FunctionInfoCollection)
                        {
                                if (fi.IsMarked)
                                {
                                        oneMarked = true;
                                        ProcessSingleFunction(reporter, fi);
                                }
                        }
                }
        }
        // If nothing is marked we will do them all
        if (!oneMarked)
        {
                foreach (IProcessInfo pi in processes)
                {
                        foreach (IThreadInfo ti in pi.Threads)
                        {
                                foreach (IFunctionInfo fi in ti.FunctionInfoCollection)
                                {
                                        ProcessSingleFunction(reporter, fi);
                                }
                        }
                }
        }
}
```

Figure 73 – C# Code representing the body of the DoMethodCallCount Entry Point

The code iterates through each of the process information objects to obtain the thread list. Once the thread list is obtained each of the thread information objects are iterated through. For each thread information object found, each function information object is then iterated through. If the function is marked, then the "ProcessSingleFunction" method is called on the marked function. If no marked functions are found in the various collections, the method will call the "ProcessSingleFunction" method on every method contained within the runtime data.

We have not yet defined the "ProcessSingleFunction" method. Add the method shown in Figure 74 to the MethodCallCount class contained within the "MethodCallCount.cs" file.

```
private void ProcessSingleFunction(IMetricReporter report, IFunctionInfo fi)
{
        string callList = "";
        foreach (ICalleeFunctionInfo c in fi.CalleeInfo)
        {
                if (c.Calls / fi.Calls == 1)
                {
                        callList += "<li>" + c.Signature + " - " + (c.Calls / fi.Calls) + " call";
                }
                else
                        callList += "<li>" + c.Signature + " - " + (c.Calls / fi.Calls) + " calls";
                callList += string.Format(" (<i>{0}% of total</i>)",
                c.PercentOfParentTimeInMethod.ToString("0.00;-0.00;0"));
                callList += "</li>";
        }
        report.ReportMetric("<Font size=-1><B>" + fi.Signature.Signature
                + "</b>:</font><br>" + string.Format("<ul><li>{0} calls</li><li>Child
                Calls:</li><ol>{1}</ol><li>{2}% of total</li></ul>", fi.Calls, callList,
                fi.PercentOfTotalTimeInMethodAndChildren.ToString("0.00;-0.00;0")));
}
```

Figure 74 – ProcessSingleFunction helper method C# listing

The ProcessSingleFunction method, determines the list of child methods that the method calls, and creates a list of each child method along with the number of times that the method is called. Finally, the callee list and the amount of time that the given method spends executing is displayed on the Contract Evaluation Report.

Step 10 – Compile and Test

That is all the code you need to create a performance extension. You should now be able to compile your extension into a .dll file named "MethodCallTest.dll". If you receive any sort of error, go back and check to make sure that you completed all of the steps correctly.

Once your can compile your extension into a .dll file, the final step is to have the Contract Evaluation Engine load and recognize your extension. To do this, copy only the "MethodCallTest.dll" file from your build folder into the "Extensions" folder which is located wherever you have installed the Contract Editor.

To see if your new extension is recognized by the Contract Evaluation Engine, start the Contract Editor and see if your new extension appears in the "Log Window" as shown in Figure 75.

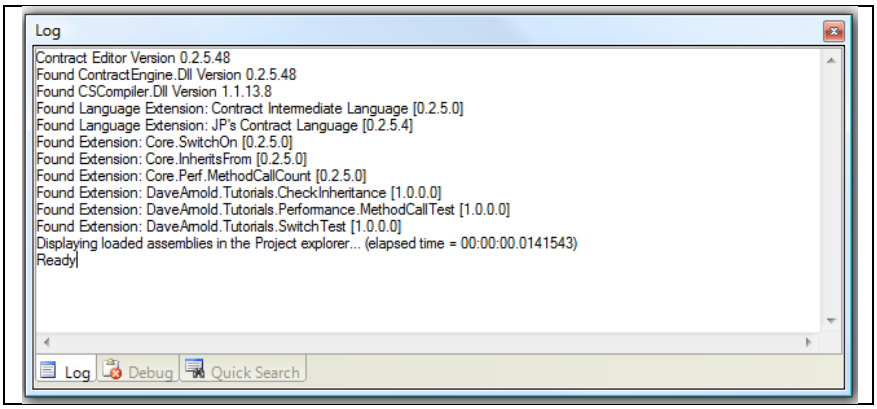


Figure 75 – MethodCallTest Performance Extension Found and Loaded

Now that your extension is found and located by the Contract Evaluation Engine, we can use it like any other extension. Using the "SwitchTest.exe" managed executable presented in the Contract Editor Overview Document [2]. We can write the JPCL contract shown in Figure 76.

```
Using DaveArnold.Tutorials.Perf;
Contract SwitchOnTest
{
        Metrics
        {
                Performance(dontcare Go(void));
        }
}
```

Figure 76 – JPCL Contract that uses MethodCallTest

The JPCL code listing creates a contract on the "SwithTestContract" type. The "SwitchTestContract" type will be bound to user supplied type via the binding tool. The contract will instruct the profiler to gather performance metrics on any method named "Go" that does not take any parameters. The JPCL "Performance" will automatically call our performance extension to analyze and report on the metric data. For more information on the JPCL syntax please see the JPCL Specification Document [5]. The JPCL compiler will transform the code shown in Figure 76, to the CIL code shown in Figure 77.

```
; CIL Generated By JPCL Compiler 0.2.5.4
; (c) Dave Arnold 2006
; Source File: SwitchOn.jpcl
; Generated On Monday, January 15, 2007 2:05:32 AM
cil             0 2 5 0
; -------------------------- Begin Contract ------------------
con             SwitchOnTest
bndr            [DaveArnold.Contracts.Examples.SwitchOn]:SwitchOn
met
pshm            ?:Go:null
cnt
; --------------------------- End Contract --------------------
econ
```

Figure 77 – CIL Contract that uses MethodCallTest

The generated CIL code is then evaluated by the Contract Evaluation Engine, which will actually call our extension and the result will be displayed in a Contract Evaluation Report, similar to the one shown in Figure 78.

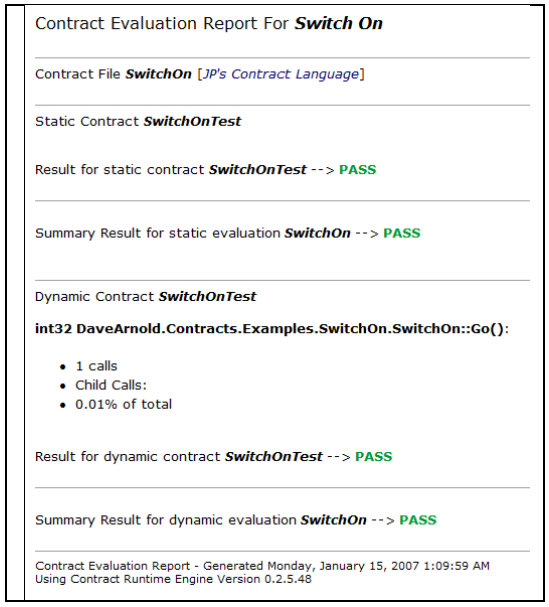


Figure 81 – Contract Evaluation Report for the Contract shown in Figure 78

The execution of a contract which makes use of our MethodCallTest performance extension concludes this tutorial.

References

[1] Arnold D., CIL Specification, Version 0.2.5.0, November 2006.

[2] Arnold D., Contract Editor Overview Document, Version 0.2.5.48, January 2007.

[3] Arnold D., Contract Language Extension Software Development Kit, Version 0.2.5.0, January 2007.

[4] Arnold D., Contract Extension Reference, Version 0.2.5.0, November 2006.

[5] Arnold D., JPCL Specification, Version 0.2.5.4, November 2006.

[6] de Icaza M., Mono C# Compiler, Version 1.1.13.8, http://www.ximian.com.

[7] Microsoft, .NET Framework 2.0 Software Development Kit, September 2005.