CHAPTER 7

The implementation of the CIL Weaving Tool

A detailed description of the implementation of the weaver tool will be given in this chapter. Section 7.1 begins with an overview of the different parts of the weaver tool. Also the data flow between the environment and the weaver tool, and the data flow between the different parts of the weaver tool are given. Section 7.2 will discuss the input supplied by the weave specifications file. This weave specification file is based on the results of the mapping of crosscutting locations to weave points, as described in section 5.3. Sections 7.3 and 7.4 are explaining the PE Weaver respectively the IL Weaver, the two major parts of the weaver tool. A short summary of the implementation of the weaver tool will be give at the end of this chapter in section 7.5.

7.1 Global structure of the weaver tool

The weaver tool consists of two major components, the PE Weaver and the IL Weaver. The PE Weaver (see section 7.3 for a detailed description) performs three steps. In the first step it converts an assembly into a textual IL file, this process is called disassembling. In the second step it calls the IL weaver to process the textual IL file. In the third, and last, step it converts the modified textual IL file to an assembly, this process is called assembling. Figure 7.1 shows this process, including the input and output of the data.

The separation of the assembling/disassembling process and the actual weaving process into two separate tools is the result of a limitation in the .NET Framework. Figure 7.1 shows that the *AssemblyInspector*, part of the IL Weaver, depends on the input assemblies. These assemblies are needed to gather information for the weaving process and are loaded into the application domain using .NET reflection. Unfortunately the .NET Framework has no method to unload an assembly from the application domain. This results in assemblies being locked by the operating system as long as the application runs. But one of the loaded assemblies may very well be an assembly we have to modify, which is not possible since it is locked (i.e. we cannot overwrite the file with the modified version). In the solution with two applications the assemblies are only loaded into the IL Weaver application domain and hence they are unlocked when the IL

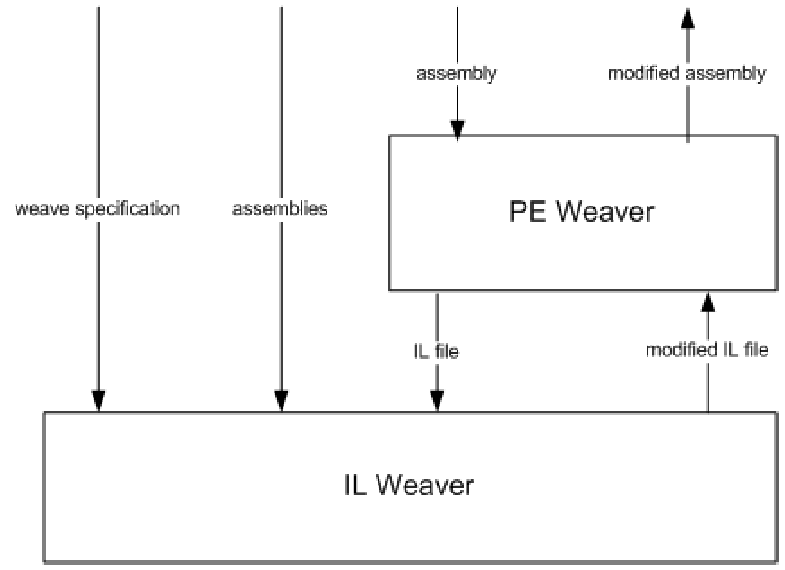

Figure 7.1: Data flow diagram weaver.

Weaver exits. Only after the IL Weaver has exited the PE Weaver will start overwriting the old assemblies with the new modified versions.

7.2 The weave specification file

To define the weave operations a separate configuration file is used, see appendix C for a complete listing of the layout of this file. This weave specification file is a xml file. Using a standard like xml has two advantages. First, the content of the file is relatively easy to understand for the developer incorporating the weaver in his own project. After all he has to make a mapping from the weave operations native to his project to the format used by the weaver. Second, using a file layout as xml allows for easy extending the functionality of the weaver. The extension of the functionality of the weaver will require additions to the weave specification file to specify this new functionality.

In listing 7.1 the general structure of the weave specification file is given. The weave specification has been given a version number to identify different versions, if this may be necessary in the future. The first block in the specification file lists references to assemblies that should be included or excluded in the assembly to be weaved, section 7.2.1 will explain this block. The second block contains all the method definitions, which can later be referenced in the application and class blocks. A more detailed description of the class block can be found in section 7.2.2. The third block specifies modifications at the application level, for example executing a method at the start of the application. Section 7.2.3 will describe the application block in more detail. The fourth, and last block specifies all the modifications at the class level. A class block is identified by the fully qualified name of the class it should be imposed on. This means that the class block is not unique, multiple class blocks with different values for the fully qualified name can occur. The details of the class block are discussed in section 7.2.4.

Listing 7.1: The weave specification file

```
<?xml version="1.0" encoding="UTF-8"?>
<weaveSpecification version="<weave specification version>">
  ["assembly reference block"]
  ["method definition block"]
  ["application block"]
  ["class block"]
</weaveSpecification>
```

7.2.1 The assembly reference block

Each assembly can contain references to other assemblies. The CLR uses these references to check if everything is available to run. When adding code, for example method calls, it can occur that the target is defined in an external assembly which was not yet referenced. For such cases the assembly reference block has the *forceReferenceIn* attribute. It can also occur that all code that uses a certain assembly is removed, making the assembly reference obsolete. The attribute *remove* provides a way to remove certain assembly references, so unused assemblies are no longer needed to pass the CLR checks. Listing 7.2 shows the structure of the assembly reference block.

Listing 7.2: The assembly reference block

```
<assemblies>
  <assembly name="<assembly name>"
            version="<assembly version>"
            publicKeytoken="<keytoken of the assembly>"
            forceReferenceIn="<assembly name>"
            remove="[yes/no]"/>
</assemblies>
```

7.2.2 The method definition block

The method definition block defines signatures of methods, which can be used in the application and class blocks. Signatures have to reflect methods that exist in the application domain. A overview of the method definition block can be found in listing 7.3.

Listing 7.3: The method definition block

```
<methods>
  <method id="<id of the method reference>"
          assembly="<assembly the method can be found in>"
          class="<fully qualified name of the class the method belongs to>"
          name="<name of the method>"
          returnType="<type of the function result>">
    <argument value="" type="[string/int]"/>
    <argument value="\%senderobject"/>
    <argument value="\%createdobject"/>
    <argument value="\%targetobject"/>
    <argument value="\%targetmethod"/>
    <argument value="\%originalparameters"/>
    <argument value="\%casttarget"/>
    <argument value="\%fieldvalue"/>
  </method>
</methods>
```

Within the method definition block more than one method definition can be defined. Each method definition consists of four attributes to identify the method and an optional number of

attributes defining the parameters of the method.
The first attribute is the *id*, this is the name this method can be referenced by in this weave specification file. The second attribute, *assembly*, contains the name of the assembly the linked method belongs to. The third attribute, *class*, defines the fully qualified name of the class this linked method is part of. The fourth attribute, *name*, is the name of the method that is being linked. Together the *assembly*, *class*, and *name* attributes uniquely identify a method.
To define the parameters of the linked method an optional number of *argument* attributes can be supplied to the method definition. There are two options to define a value for a parameter: a constant value, or a dynamic value based on the runtime context when the method is called. Currently the only constant values supported are the string and integer types. The currently supported dynamic values are explained in more detail below:

%senderobject (type *object*): The object in which the weave point is found is passed to the target function. Note that this has only any meaning in a non-static context, i.e. the weave point is found in a non-static context.

%createdobject (type *object*): This value can only be used in the context of a *after class instantiation block* (see below for a description). At runtime the newly created object instance will be passed as parameter of the target function. The parameter type for this parameter of the target function is the *object* type. There are two reasons to use the most basic type, *object*, for this. First, the target function can now be used in more than one class block. This allows the developer to create a single function for handling the class instantiation, for example logging the creation of new objects with the type of the object as extra information. Second, it allows the developer to seperate the application from the methods handling the weave points. Which in turn allows easy re-use of weave point handling methods by putting them in a seperate assembly, independant of the application.

%targetobject (type *object*): If the weave point is a non-static operation on an object, e.g. a method call, the object can be passed to the target function.

%targetmethod (type *string*): If the weave point is a method call the name of the method can be passed to the target function using the *%targetmethod* value. This can be used when the weave point is very general, for example all calls to methods of a certain object. The target function can now be supplied with the exact name of the called method intercepted at that point.

%originalparameters (type *object array*): In case the weave point is a method call, the *%originalparameters* value can be used to pass all the parameters of the original method call to the target function. The original parameters are placed in an array and are passed as a single parameter to the target function.

%casttarget (type *string*): If the weave point is the cast from one object type to another, the target function can be supplied with the object being casted and the type casted to. The object being casted can be passed to the target function using the *%targetobject* value. To pass the type casted to the *%casttarget* value can be used. The type of this value is the *string* type, in other words the target function receives the name of the type being casted to.

%fieldvalue (any type): If the weave point is an operation on a field, the original value of the field can be passed to the target function with the *%fieldvalue* value.

7.2.3 The application block

The structure of the application block is given in listing 7.4. Below is a explanation of the different attributes.

As identified in chapter 5 (see the dynamic crosscutting locations in table 5.3) the start of the application is a weave point. The application block allows to link a method, defined in the *method definition block*, to the start of the application using the *notifyStart* attribute.

Listing 7.4: The application block

```
<application name="">
  <notifyStart id="<reference to a method definition>"/>
</application>
```

7.2.4 The class block

In listing 7.5 the layout of the class block is given. A weave specification file can contain multiple class blocks, each identified by a different class name in the *name* attribute. Apart from identifying a class block by the *fully qualified class name*, a wild card can be used as *class name*. A wild card matches every class in the application. For example this can be used to add a call to a logging method in every class at a file operation.

Listing 7.5: The class block

```
<class name="[*/<fully qualified class name>]">
  ["after class instantiation block"]
  ["method invocation block"]
  ["cast block"]
  ["class replacement block"]
  ["field access block"]
</class>
```

The class block contains four sub blocks, which will be explained below.

The after class instantiation sub block

The *after class instantiation block* contains a method reference in the *executeMethod* part. The id of the reference method is stated in the *id* attribute and refers to a previously defined method in the *method definition block* (see section 7.2.2). Listing 7.6 shows the *after class instantiation block*. This weave point is implemented in the method that instantiates the class, i.e. after a new object of the class has been created. Note that this allows us to pass the newly created object as parameter (using the %createdobject value) to the referenced method.

Listing 7.6: The after class instantiation block

```
<afterClassInstantiation>
  <executeMethod id="<reference to a method definition>"/>
</afterClassInstantiation>
```

The method invocation sub block

As identified in chapter 5 the call to a method is a weave point. This weave point can be specified in the *method invocation block*, see listing 7.7. A *methodInvocations* node can contain multiple *callToMethod* sub nodes, each identified by a different fully qualified method name (the *class* and *name* attributes). Contained in the *callToMethod* node are the *voidRedirectTo* and *returnvalueRedirectTo* nodes. These nodes are used to specify the target method to call, identified by its *id* attribute.

Listing 7.7: The method invocation block

```
<methodInvocations>
  <callToMethod class="" name="">
    <voidRedirectTo id="<reference to a method definition>"/>
    <returnvalueRedirectTo id="<reference to a method definition>"/>
  </callToMethod>
</methodInvocations>
```

The reason for two nodes to identify the target method is due to the fact that a .NET method can have a return value. If the intercepted method does not have a return value, in other words has the return type *void*, the target method reference in the *voidRedirectTo* node will be used. Otherwise if the intercepted method has a return value the target method reference in the *returnvalueRedirectTo* will be used. As return type of the target method the *object* type can be used. The weaver will automatically cast it to the expected return type. Note that it is the responsibility of the target method to return a correct castable type.

The cast sub block

In listing 7.8 the layout of the *cast block* shown. The weave point identified by the *cast block* is the cast from one class to another class as supported by the class hierarchy of the .NET Framework. Matching will occur on the fully qualified name of the target class of the cast, defined with the *assembly* and *class* attributes of the *castTo* node. The only supported action at this weave point is the execution of a target method. A reference to this method is stated in the *id* attribute of the *executeMethodBefore* node.

Listing 7.8: The cast block

```
<casts>
  <castTo assembly="" class="">
    <executeMethodBefore id="<reference to a method definition>"/>
  </castTo>
</casts>
```

Passing the original object, i.e. the object being cast, to the target function can be done by defining the *%casttarget* value for the target function. The target method is expected to have a *object* as return type, logically it should be castable to the expected target type. This approach allows the target function to modify the original object, which can be exploited to introduce multi-inheritance into the single-inheritance class hierarchy of the .NET Framework. In the .NET code the type checking on casting can be avoided by explicitly casting to the *object* type before the desired casting is done, e.g. *TypeA a = (TypeA)(Object)(TypeB)*. At runtime this code will give a casting exception, but the target function can now substitute a valid object to allow multi-inheritance. Note that the entire mechanism to keep track of a multi-inheritance tree and provide the right object at every weave point is not provided by this weaving tool.

The class replacement sub block

The *class replacement block* is in fact a simple renaming operation. See listing 7.9 for the structure of the *class replacement block*. Bound to the parent class block operations performed on a certain class can be redirected to another class. Inside the *class replacement block* multiple replacements can be defined, i.e. it can contain more than one *classReplacement* node. The class to be replaced (or renamed) is identified by the *assembly* and *class* attributes of the *classReplacement* node. The name of the target assembly and class are defined respectively in the *assembly* and the *class* attributes of the *replaceWith* node inside the *classReplacement* node. For example logging operations in an existing application can be redirected to a new logging class as long as the public interface of the new logging class matches the interface in the existing application.

Listing 7.9: The class replacement block

```xml
<classReplacements>
  <classReplacement assembly="" class="">
    <replaceWith assembly="" class=""/>
  </classReplacement>
</classReplacements>
```

The field block

Three different operation can be identified at the field access weave point. Field access is defined as a reading or writing the value of a field. The three operations are: calling a target funtion before the field is accesed, calling the target function after the field is accesed, and replacing the field access operation with a call to the target function. An overview of the field access block can be found in listing 7.10.

Listing 7.10: The field access block

```xml
<fieldAccesses>
  <field class="" name="">
    <callBefore id="<reference to a method definition>"/>
    <callAfter id="<reference to a method definition>"/>
    <replaceWith id="<reference to a method definition>"/>
  </field>
</fieldAccesses>
```

The *class* and *name* attributes of the *class* node are used to identify the fully qualified name of the field. Note that the *fieldAccesses* node can contain multiple *field* nodes for different fields. The method referenced by a *replaceWith* node has to have its *returnType* attribute defined. It is the responsibility of this target function to return a valid type, i.e. the type of the result should not cause a .NET type conflict with the original type of the field.

7.3 The PE Weaver

As shown in figure 7.1 the main tasks of the PE Weaver are disassembling the original assemblies and assembling the modified assemblies. But apart from these two main tasks the PE Weaver can do verification of the original and modified assemblies. A more detailed view of the PE Weaver can be found in figure 7.2.

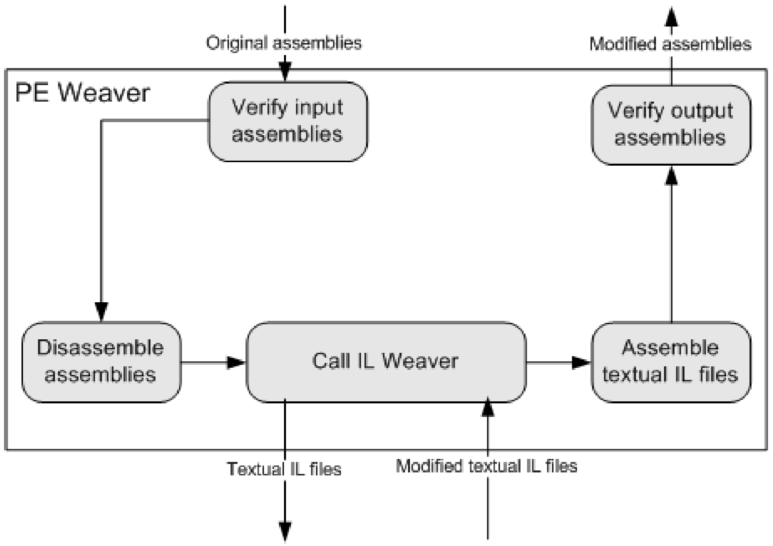

Figure 7.2: Data flow diagram PE Weaver.

7.3.1 Verification of the assemblies

To verify the assemblies before and after the weave process the *peverify tool* distributed with the .NET SDK is used. This tool checks the assembly for various errors, e.g. stack overflows or unknown method/assembly references. Especially in the development process of a tool using the weaver error checking can be very useful.

7.3.2 Disassembling

Disassembling an original assembly is one of the main tasks of the PE Weaver. To disassemble an assembly the *ildasm tool* (IL Disassembler) distributed with the .NET SDK is used. The output of the *ildasm tool* is a text file with the textual representation of the IL code in the assembly. This textual IL file is used as input for the IL Weaver, see section 7.4.

7.3.3 Assembling

After the IL Weaver has modified the textual IL files a new assembly has to be created. This process is called *assembling* and is done by the *ilasm tool* (IL Assembler) part of the standard .NET Distribution. If the assembling succeeded without errors the original assembly will be overwritten, unless the special */out* command-line switch of the PE Weaver is used. With this switch it is possible to define the name of the new assembly and the original assembly will not be overwritten. This option is only usable when weaving a single assembly.

7.4 The IL Weaver

The IL Weaver performs the actual weaving of the instructions from the weave specification file into the IL code. Figure 7.3 shows the main tasks performed by the IL Weaver.

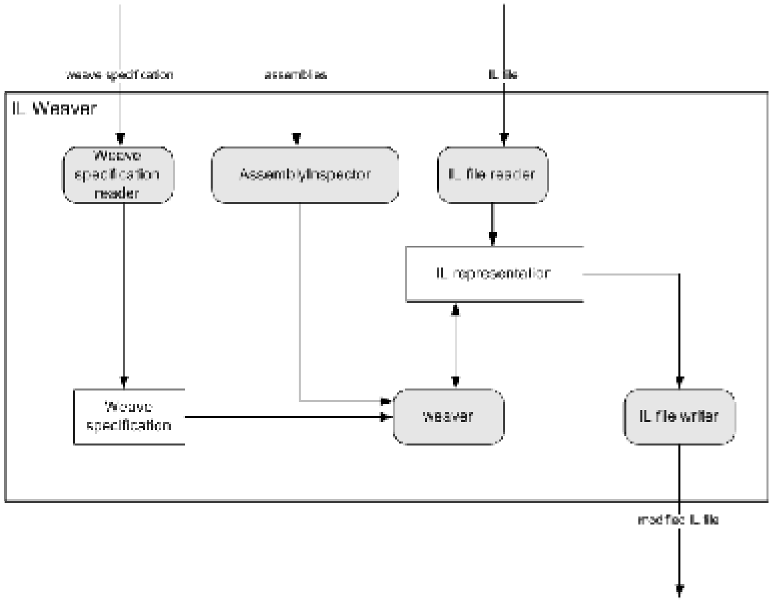

Figure 7.3: Data flow diagram IL Weaver.

The input for the IL Weaver is the weave specification file, the textual IL file(s) containing the code (provided by the PE Weaver), and all assemblies needed to extract additional information from about the .NET structure of the application. The output from the IL Weaver is a modified version of the textual IL file(s).
Additional information about these steps performed by the IL Weaver can be found below.

7.4.1 Reading the weave specification file

In section 7.2 the detail of the weave specification file have been explained. To use the information stored in the weave specification file the IL Weaver has to read it and store the information into an internal structure. This internal structure is needed to query the information fast.
The IL structure is located in the *namespace Weavers.IlStructures* namespace and contains the following classes: NamespaceBlock, ExternalAssemblyBlock, ClassBlock, MethodBlock, and IlOpcode.

7.4.2 Reading a IL file

Before the actual weaving can be done the textual IL file generated by the PE Weaver has to be loaded. To be able to reason about the structure of the IL code contained in the textual IL file, the file is parsed into a special structure. This structure could in fact be a complete representation of the CIL language, but we haven chosen to only implement the part of the structure we need.

We have chosen for this approach for the following reasons: simplicity, less error prone and upgradability. The first reason, simplicity, is because the CIL specifications is very detailed and complex. Besides implementing a structure to store every possible CIL option is not necessary, we only need a very limited part for the weave process. The second reason, less error prone, also results from the complexity of the CIL specifications. If we limit the structure to what we need and do not try to parse the rest, we only have to worry about a very small part of the complete CIL specifications. The third and last reason, upgradability, means that changes to the CIL specifications are less likely to have impact on the structure used to store the IL code.
A complete class diagram of the internal IL structure can be found in appendix D.

7.4.3 The assembly inspector

The assembly inspector has the task to gather information about types in the application domain, when needed to perform certain weave operations. For example checking inheritance relationships. The best way to get this information is using .NET Reflection. To be able to query information about a certain assembly using reflection the assembly has to be loaded into the application domain of the IL Weaver. To obtain the best performance the assembly inspector uses an internal hashtable to store results, at the cost of increased memory usage. To keep the results available for the weave process of every IL file the assembly inspector is implemented as a singleton.
Listing 7.11 shows the public interface of the assembly inspector. This *isMethod* method will iterate over the list of assemblies supplied by the *enumAssemblies* parameter and look for the class or type stated in the *className* parameter. If this type is found it is retrieved with reflection using the *GetType* method. The result is stored in a *System.Type* object. The final step is using the *GetMember* method of the *System.Type* object to determine if the supplied *methodName* parameter is an existing method of this type.

Listing 7.11: Public method *IsMethod* of the assembly inspector

```
public bool IsMethod(String currentAssembly,
                     IEnumerator enumAssemblies,
                     String className,
                     String methodName)
```

Especially in the process of determining the class hierarchy for casting operations this functionality is used. Because the weave specifications are defining a certain class to weave on, but in the case of methods inherited from a parent class the IL code states the parent class. Matching just the class from the weave specifications to the class name in the IL code will not give the proper result.

7.4.4 Weaving

In the weave process an iteration over all the IL structures, based on the textual IL file, is performed. Inside the *MethodBlock* class the IL instructions are stored in an array of *IlOpcode* objects. Comparing the value of an *IlOpcode* object with the supported weave points will give the weave points in the IL code. When a weave point is identified a weave operation can be performed, for example calling the defined target function.
After all the weave points in a *MethodBlock* have been found the weave process will update the IL administration of the method. In a number of cases the addition of calls to a target function required the increase of the maximum stack size value of the method.

7.5 Summary

In this chapter the implementation of the CIL Weaver has been discussed. The CIL Weaver is a combination of two tools: the PE Weaver, and the IL Weaver. The PE Weaver disassembles the input assemblies into textual IL files. These textual IL files are used as input for the IL Weaver. After the IL weaver finishes the weaving, the PE Weaver assembles the textual IL files into new assemblies. The IL Weaver performs the actual weaving and used the following input: the weave specification, the textual IL files, and related assemblies to gather additional information. As output the IL Weaver gives a modified textual IL file, which can be assembled by the PE Weaver.
As stated earlier one of the goals of the CIL Weaver is providing a tool other developers can use to implement their own AOP tool. In the next chapter (chapter 8, Integrating the CIL Weaver into Composestar) the integration of the CIL Weaver into the Composestar project is described. In a wider context this can be seen as an example of integrating the CIL Weaver into an AOP solution.