Masarykova univerzita
Fakulta informatiky



Design architecture and implementation of development environment for ATML (Automated Test Mark-Up Language)

Bachelor thesis

Michal Sorentíny

Brno, spring 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Michal Sorentíny

Advisor: prof. Ing. Václav Přenosil, CSc.

Acknowledgement

I would like to thank my supervisor... prof. Ing. Václav Přenosil, CSc. who gave me valuable advices and to my technical supervisor Ing. Miloš Machat for a professional lead with implementing the ATML Tools software.

Abstract

The aim of the bachelor work is to provide an overview of implementing software for creating XML files compliant to IEEE Std 1671. The work compares tools for automatic generating C# source codes from XML Schema files and describes problems with tools available on market and how to overcome them.

The work also gives insight into extensible software architecture, describes .NET frameworks providing extensibility options - MEF and MAF.

Part of the work consists of design and implementation of graphical user interface for the ATML Tools software.

The result product - ATML Tools is used in Honeywell corporation for commercial purposes, which is a reason why only screen shots of the ATML Tools application and source codes made for this thesis are provided.

Keywords

ATML, IEEE Std 1671, C#, .NET, Managed Extensibility Framework, XSD2Code, GUI

Contents

1 **Introduction** . . . 1
2 **The general problem analysis** . . . 2
2.1 *General problem solutions* . . . 4
2.2 *Goals of the work* . . . 4
3 **Converting XML schema structures into C# classes** . . . 6
3.1 *Problem Analysis* . . . 6
3.2 *Possible solutions* . . . 6
3.2.1 Manual writing of objects . . . 7
3.2.2 XSD Schema Definition Tool (Xsd.exe) . . . 8
3.2.3 LINQ to XSD . . . 11
3.2.4 XSD2Code . . . 12
3.3 *Conclusion* . . . 13
3.4 *Generating classes with XSD2Code* . . . 15
3.4.1 Custom changes in XSD2Code tool . . . 18
4 **ATML Tools modular architecture** . . . 21
4.1 *Problem analysis* . . . 21
4.2 *Possible solutions* . . . 22
4.3 *Conclusion* . . . 25
4.4 *Solution* . . . 26
5 **GUI design** . . . 28
5.1 *Problem analysis* . . . 28
5.2 *Solution* . . . 29
5.2.1 Main window . . . 29
5.2.2 Editor window . . . 30
6 **Conclusion** . . . 32
6.1 *Further improvements* . . . 32
A **ATML Tools screen shots** . . . 35
B **CD content** . . . 38

1 Introduction

"For many years, the ATE (Automatic Test Equipment) industry has coveted a universal data exchange adapter for ATE, spurred by the need to share test system and test result information with the rest of the enterprise. Many test organizations within the ATE industry have already developed or are considering developing their own proprietary XML-based data exchange standards to meet their specific needs for sharing ATE and test information. However, this may no longer be necessary with ATML."[1]

This whole work describes my work as a test software engineer on the "ATML Tools" product being developed by Honeywell company. Due to this fact, the application's source codes can not be publicly available and the appendices contain either only parts of the actual source codes or examples made by myself specifically for this bachelor thesis and screen shots of the actual application.

The first chapter of this thesis analyzes the problem and introduces the Automatic Test markup Language ATML technology.

The second chapter provides an overview of tools usable for automatic source code generation from XML Schema files and justifies our choice of the specific tool.

The third thesis chapter describes a software architecture of an ATML Tools and shows aspects and results of graphical user interface design.

Fourth chapter collects requirements and guidelines for graphical user interface of the ATML Tools and delivers sample screen shots from the actual application.

2 The general problem analysis

For the automatic tests it is essential to be able to configure the tests and store the results in a format which should be both easily processed by computers and configurable by a human without spending a big effort. The XML (Extensible Markup Language) satisfies both conditions.

XML

XML belongs to a group of markup languages. People started using markup languages a few centuries ago. They were used at traditional publications and marking up manuscripts by adding instructions for a printer in the margins of manuscripts. He used these markups to know which font was about to be used and then passed the manuscript for typesetting. As a “father” of the modern markup languages is considered to be Charles Goldfarb, an IBM researcher, who transformed the idea into a fully functional product. Nowadays, the XML has widely spread to make a data exchange easier in the field of information technologies. This meta markup language describes rules how to define your own set of markups and ability to interpret them clearly in the future. Number of tools for parsing and processing XML files have been developed over many years. However, the XML is only a meta-language, so the set of elements used for describing the tests can vary not only from company to company, but also inside one corporation. That is why the IEEE Standard Association (IEEE-SA) developed the international standard IEEE Std 1671™ which was published as “IEEE Standard for Automatic Test Markup Language (ATML) for exchanging Automatic Test Equipment and Test Information via XML”.

ATML

"Automatic test markup language (ATML) is a collection of IEEE standards and associated extensible markup language (XML) schemas that allows automatic test system (ATS) and test information to be exchanged in a common format adhering to the XML standard."[2]

Historical Background

"In 2002, an automatic test markup language (ATML) focus group was formed (outside any formal standardization body) with a mission to "define a collection of XML [extensible markup language] schemas

that allows ATE [automatic test equipment] and test information to be exchanged in a common format adhering to the XML standard." The scope of this effort was the standardization of test information, which would allow for test program (TP) and test asset interoperability, as well as unit under test (UUT) data (including results and diagnostics), to be interchanged between heterogeneous ATE systems."[2]

ATML is an XML-based standard for ATE and test information data exchange. It supports describing of wide spread of technical data and from simple electrical signal description to complex test stations. It also enables to define behavior of individual components and information exchange in a uniform format independent on used platform and hardware. Because ATML uses the XML standard for describing ATE and test data, it takes advantage of recursion and extensibility, key technical benefits that give test systems greater flexibility in defining interfaces. By supporting recursion, element definitions describing test properties or tests can nest to create a more managed data exchange format. The IEEE-1671 standard consists of 6 parts covering different sections. A brief overview of the ATML Standard is:

- 1671.1 covers test descriptions, i.e. the descriptions of UUTs (units under test) and the particular set of tests that have to be performed on them.
- 1671.2 covers instrument descriptions and specific instrument instances, i.e. descriptions of general and specific instruments, which will often become UUTs.
- 1671.3 covers general UUT descriptions and specific UUT instance descriptions.
- 1671.4 covers test configurations.
- 1671.5 covers test adapter descriptions.
- 1671.6 covers descriptions of test stations.

In a short way, the problem is to find an effective way how to create documents compliant with the very complex XSD schemas of the IEEE-1671 standard.

The aim of the work itself is to provide a detailed design of software development tools which shall create XML files compliant with IEEE-1671 standard. It covers my work as a designer of the main architecture, the choice of the used transformation technologies between XSD schema files and C# classes and a part of the GUI design. The application development pays attention to design the ATML Tools modular, so that written code can be massively reused across the different versions of software for different IEEE-1671 standard versions, which are not compatible with each other.

2.1 General problem solutions

Since the ATML standard is relatively new and most of it is still in the trial-use state there are not many solutions implementing the ATML in the test systems. According to Ron Taylor, one of the main impediments implementing ATML is "Lack of ATML tools commercially available – The lack of ATML development tools commercially available is an issue that is limiting the use of ATML. [...] In addition as new programs are beginning to require ATML, the demand for tools will increase and drive their development. Without ATML tools the instance files must be generated manually which is time consuming and error-prone."[6] Therefore the Honeywell ETS (Engineering Test Services) team based in Brno, Slatina decided to implement the ATML standard into their test processes and use it in their CTS (Common Test Station) as the format of exchanging test results. Implementing the ATML in the CTS includes development of the ATML Tools software in the .NET 4.0 framework.

2.2 Goals of the work

- Process the final work on the topic Detailed design of development tools for ATML IEEE Std 1671 standard-compliant XML files.
- Find out possible ways of automated conversion of complex XML structures (defined by XML schemas) into C#/.NET data structures or classes.

- Collect and prepare common requirements and guidelines for ATML tools graphical user interface.
- Implement a part of ATML development tools GUI.

3 Converting XML schema structures into C# classes

This chapter will describe the technologies used for making us able to work with the structures and types defined in the ATML standard. The goal was to achieve this functionality in the simplest possible way with the minimal effort needed. I will describe a comparison of possible technologies and attach short examples of the generated source codes as well as examples of working with the generated source code.

3.1 Problem Analysis

Since the ATML standard is in fact a set of XML schema definition (XSD) files containing a big amount of more or less complex structures and types there is the need of the conversion of these types to C# classes. These classes will represent each of the ATML XSD complex types or simple types. Instances of these classes will be edited in the desktop application and then serialized back to XML and saved on disc.

3.2 Possible solutions

Few possibilities emerged during solving this problem:

a) Manual writing of objects

b) Automatic conversion using XML Schema Definition Tool (Xsd.exe)[1]

c) Automatic conversion using LINQ to XSD[2]

d) Automatic conversion using XSD2Code[3]

---

1. http://msdn.microsoft.com/en-us/library/x6c1kb0s(v=vs.100).aspx
2. http://linqtoxsd.codeplex.com/
3. http://xsd2code.codeplex.com/

The tool for automatic generation of classes shall provide the following features:

- tool shall be free for commercial use
- generate simple, readable and understandable source code
- the generated code should not need any manual adjustments (adding code for serializing and deserializing objects or for input - output operations)
- the source code should contain comments generated from XSD annotations
- generate types from imported schemas into separate files
- the tool shall be maintained by an external provider
- tool should be integrated into the Microsoft Visual Studio 2010, if possible

Examples demonstrating Load from file and Save to file operations are provided below for solutions b), c) and d). As a base XML schema for generating examples I have used the Instrument Description[4] schema. The source code comments in the examples are omitted in the printed version and can be viewed in the appendix.

3.2.1 Manual writing of objects

Since the number of types and their complexity in the ATML standard is significant, the manual writing of the classes has been rejected immediately and as the only option proved to be the automatic generation of the classes. The manual writing had not been considered as usable for the ATML Tools project purposes and it won't be considered in the thesis or figures anymore.

---

4. InstrumentDescription.xsd, available from `http://grouper.ieee.org/groups/scc20/ATML/Demonstrations/Phase3/Baseline%20ATML%20Schemas/InstrumentDescription.xsd`

3.2.2 XSD Schema Definition Tool (Xsd.exe)

The XSD Schema Definition Tool was the first tool considered to be used for the automatic code generation purposes. It is the part of the .NET framework since .NET 1.1 and it communicates with user via command line. This tool has been considered as the first possibility of generating the classes we needed, since it is it's main purpose. The tool did it's work, however it does not provide us the settings and the configuration possibilities necessary for our purposes - excluding imported types, generating source code comments and it's overall configurability is not satisfactory. Moreover, considerable amount of code had to be added for every generated class to be able to perform basic operations like loading and saving into the XML files. Example of serialization, deserialization and load - save source code needed to be added:

```
1  public static InstrumentDescription Deserialize(
       string xml)
2  {
3      System.Xml.Serialization.XmlSerializer
           serializer = new System.Xml.Serialization.
           XmlSerializer(typeof(InstrumentDescription)
           );
4
5      System.IO.StringReader stringReader = null;
6      try
7      {
8          stringReader = new System.IO.StringReader(
               xml);
9          return ((InstrumentDescription)(serializer
               .Deserialize(System.Xml.XmlReader.
               Create(stringReader))));
10     }
11     finally
12     {
13         if ((stringReader != null))
14         {
15             stringReader.Dispose();
16         }
```

```
17      }
18  }
19
20  public static InstrumentDescription LoadFromFile(
        string fileName)
21  {
22      System.IO.FileStream file = null;
23      System.IO.StreamReader sr = null;
24      try
25      {
26          file = new System.IO.FileStream(fileName,
                FileMode.Open, FileAccess.Read);
27          sr = new System.IO.StreamReader(file);
28          string xmlString = sr.ReadToEnd();
29          sr.Close();
30          file.Close();
31          return Deserialize(xmlString);
32      }
33      finally
34      {
35          if ((file != null))
36          {
37          file.Dispose();
38          }
39          if ((sr != null))
40          {
41              sr.Dispose();
42          }
43      }
44  }
45
46  public static string Serialize(
        InstrumentDescription instrumentToSerialize)
47  {
48      System.Xml.Serialization.XmlSerializer
            serializer = new System.Xml.Serialization.
            XmlSerializer(typeof(InstrumentDescription)
```

```
        );
49      System.IO.StreamReader streamReader = null;
50      System.IO.MemoryStream memoryStream = null;
51      try
52      {
53          memoryStream = new System.IO.MemoryStream
                ();
54          serializer.Serialize(memoryStream,
                instrumentToSerialize);
55          memoryStream.Seek(0, System.IO.SeekOrigin.
                Begin);
56          streamReader = new System.IO.StreamReader(
                memoryStream);
57          return streamReader.ReadToEnd();
58      }
59      finally
60      {
61          if ((streamReader != null))
62          {
63              streamReader.Dispose();
64          }
65          if ((memoryStream != null))
66          {
67              memoryStream.Dispose();
68           }
69       }
70  }
71
72  public static void SaveToFile(
        InstrumentDescription instrument, string
        fileName)
73  {
74      System.IO.StreamWriter streamWriter = null;
75      try
76      {
77          string xmlString = Serialize(instrument);
```

```
78          System.IO.FileInfo xmlFile = new System.IO
               .FileInfo(fileName);
79          streamWriter = xmlFile.CreateText();
80          streamWriter.WriteLine(xmlString);
81          streamWriter.Close();
82      }
83      finally
84      {
85          if ((streamWriter != null))
86          {
87              streamWriter.Dispose();
88          }
89      }
90  }
91
92  public static void XSDexeLoadSaveDemo()
93  {
94      // Load Instrument Description from PXI.xml
           file
95      InstrumentDescription instrument =
           LoadFromFile("PXI.xml");
96
97      // Save Instrument Description to PXI.xml file
98      SaveToFile(instrument, "PXI1417.xml");
99  }
```

The Xsd.exe tool does not provide any options to generate code comments from XSD annotations, which is necessary since the ATML standard contains hundreds of types. After an additional research, this tool appeared to be used in the XSD2Code application, which will be described later.

3.2.3 LINQ to XSD

The first application which was considered as the fully usable for our purpose was an application which contributes to the LINQ (Language Integrated Query) project, in particular, enhances the existing LINQ to XML technology. LINQ to XSD is released under the Microsoft Public

License (Ms-PL), as the user interface uses the command line and in the newest version needs at least .NET framework 3.5.

Compared to Xsd.exe, LINQ to XSD generates all methods needed for serialization and deserialization inside every generated class. The Load and Save to file source code is therefore very simple:

```
1 static void LinqToXSDLoadSaveDemo()
2 {
3     // Load Instrument Description from PXI.xml
4     InstrumentDescription instrument =
          InstrumentDescription.Load("PXI.xml");
5
6     // Save instrument Description to PXI1447.xml
7     instrument.Save("PXI1447.xml");
8 }
```

Despite LINQ to XSD generates source code comments, it does suffer from the same issue as the Xsd.exe - it can not exclude types included from other schemas from generation and provides the lack of options, this tool became unusable. Unfortunately, this project is not maintained[5] since the 2010 and because it suffers from some serious issues crucial for ATML Tools project, it was decided not be used for our purposes.

3.2.4 XSD2Code

The last tool we worked with was the XSD2Code application, released under Microsoft Reciprocal License (Ms-RL). It is available both as a command line application and Visual Studio extension, where provides Graphical User Interface (GUI) for easier configuration (Fig. 1).The application wraps the code generated by the Xsd.exe tool mentioned above with code needed for serializing and deserializing each of generated types, saving and loading from file and makes the user able to choose the type of collections used inside. The available collections are: `Array`, `List<T>`, `IList<T>`, `ObservableCollection<T>` and defined types. XSD2Code also provides the user with an option to generate source code comments

5. http://linqtoxsd.codeplex.com/discussions/81738

from XSD annotations. The source code needed to load and save the InstrumentDescription object is similar to LINQ to XSD's:

```
1 static void Xsd2CodeLoadSaveDemo()
2 {
3     // Load InstrumentDescription from PXI.xml
        file
4     InstrumentDescription instrument =
        InstrumentDescription.LoadFromFile("PXI.xml
        ");
5
6     // Save InstrumentDescription to PXI1442.xml
7     instrument.SaveToFile("PXI1442.xml");
8  }
```

Automatic updates of the classes after updating the XSD files in the Visual Studio project is a welcomed function as well, but rather unused in the ATML Tools. The XSD2Code has an alive community of contributors and is continuously developed. In general, the XSD2Code provided the best configuration options and fulfilled the most of our requirements for our needs.

3.3 Conclusion

To abide Honeywell Quality Function Deployment (QFD) process requirements, the following decision matrix 3.2 along with a table of few pros and cons 3.3 has been used. The decision matrix is an easy to understand model. Each criterion has an importance ranking and a score in range of 0 to 10. Criterion score is computed by multiplying these two values. Total score is made by an addition of total criterion scores.

After the QFD decision matrix results along with the additional criteria (additional source code needed to work with generated classes) were considered the XSD2Code tool has been chosen as the proper tool for our purposes with the most pros and none of the negative features.

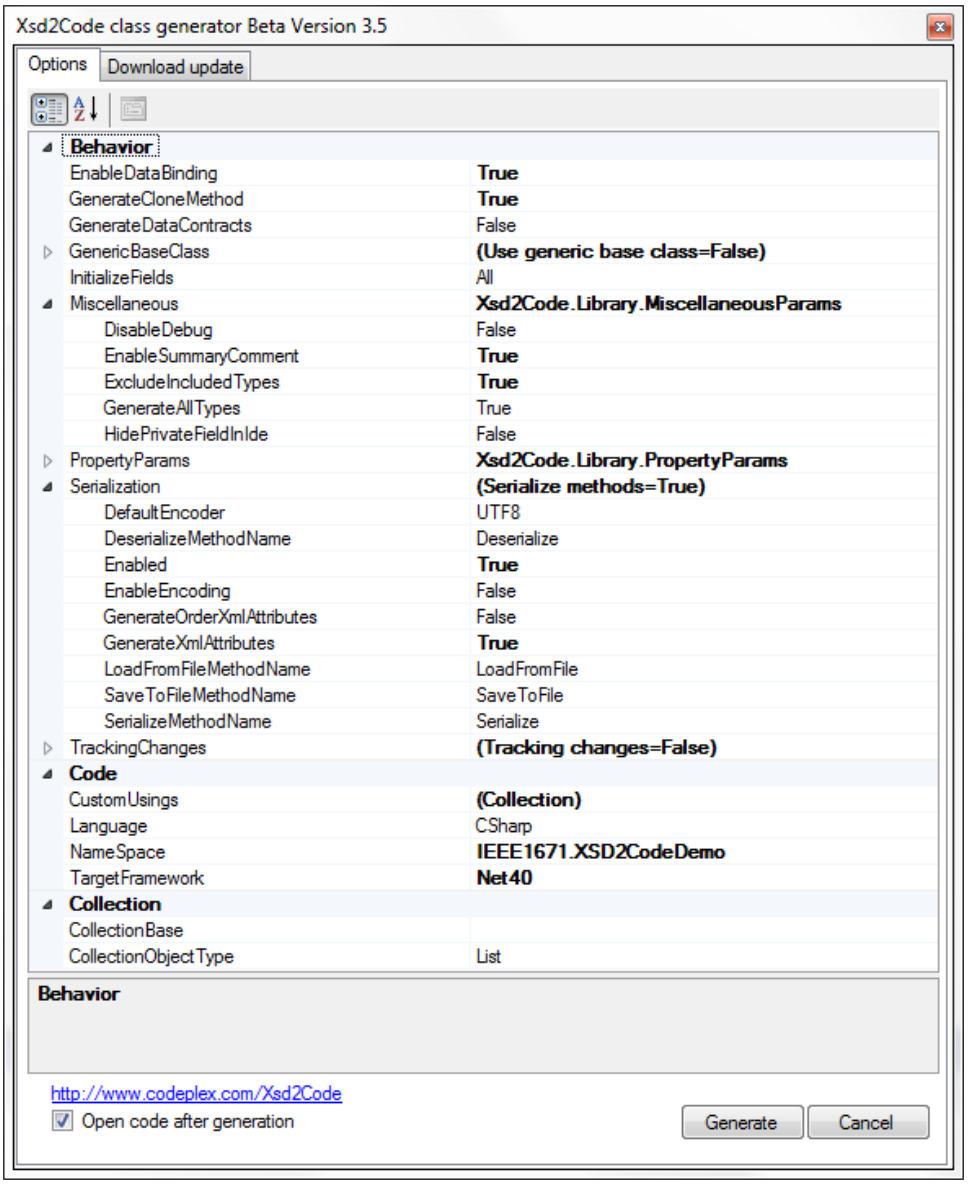


Figure 3.1: XSD2Code GUI, taken from the XSD2CODE application

QFD		Requirement / Features (rule / approach)	XML Schema Definition Tool		LINQ to XSD		XSD2Code	
#	Customer Needs	Imprtnc Rank'g						
1	Configurability as high as possible	30	3	90	7	210	10	300
2	Development tools integration as high as possible	10	5	50	5	50	10	100
3	Software maintained by an external provider as high as possible	30	5	150	3	90	7	210
4	Generated code separability as high as possible	30	5	150	5	150	7	210
		Total	440		500		820	
		Relative Ranking	3		2		1	

Figure 3.2: Automatic code generation QFD

	XML Schema Definition Tool	LINQ to XSD	XSD2Code
Pros	Integrated into .NET Framework	Plugins can run in their own app domain	Allows generation of separated classes
			Live open-source project
			Integrated into Visual Studio
Cons	Poor configurability, does not generate source code comments	Project is dead since November 2010	-

Figure 3.3: Automatic code generation tools pros and cons

3.4 Generating classes with XSD2Code

As the decision has been made, we started to generate the ATML types from the XSD schemas using the XSD2Code tool. After reading

documentation and several trials to use the generated code, we used a following options:

- Serialization Enabled - True
- IncludeCloneMethod - True
- GenerateXmlAttributes - True
- EnableDataBindings - True
- ExcludeIncludedTypes - True
- Platform - .NET 4
- CollectionBase - Array
- EnableSummaryComment - True
- Namespace - "`IEEE1671_20xx.yyyyy`" (according to ATML version and generated type - e.g. `IEEE1671_2011.TestStationInstance`)
- CustomUsings - every ATML type that includes types from Common.xsd[6] and HardwareCommon.xsd[7] has added include of `IEEE1671_20xx.Common` and `IEEE1671_20xx.HardwareCommon` generated separately.

The other options were set to default. After examining the generated source code, we found out it had a serious problem. If the XSD schema contains an attribute group and another attribute group which contains referenco to the first one, the generated type containing the extended attribute group does not contain the nested group's attributes. Sample XSD source code taken from the actual Common.xsd and InstrumentDescription.xsd without annotations is provided for better understanding the problem:

```
1 <xs:attributeGroup name="ClassifiedAttributes">
2 <xs:attribute name="classified" type="xs:boolean"
    use="optional" />
3 <xs:attribute name="securityClassification" type="
    c:NonBlankString" use="optional" />
```

```
</xs:attributeGroup>

<xs:attributeGroup name="DocumentRootAttributes">
<xs:attribute name="uuid" type="c:Uuid" use="
   required" />
<xs:attributeGroup ref="c:ClassifiedAttributes"/>
        </xs:attributeGroup>
```

```
<xs:complexType name="InstrumentDescription">
  <xs:complexContent>
    <xs:extension base="hc:HardwareItemDescription
       ">
      <xs:sequence>
        [...]
      </xs:sequence>
    <xs:attribute name="type" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Instrument"/>
        <xs:enumeration value="Module"/>
        <xs:enumeration value="Option"/>
      </xs:restriction>
    </xs:simpleType>
    </xs:attribute>
    <xs:attributeGroup ref="
       c:DocumentRootAttributes"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

From the XML schema provided above it is obvious, that the InstrumentDescription object contains all attributes specified in the `DocumentRootAttributes` attribute group, which means also the attributes classified and `securityClassification` inherited from the ClassifiedAttributes attribute group. However, the generated InstrumentDescription object contained only the attributes specified directly in the InstrumentDescription complex type - a required attribute named type and the non-inherited attribute specified in the `DocumentRootAttributes` attribute group - a required attribute

named `uuid`.

3.4.1 Custom changes in XSD2Code tool

To solve the serious problem mentioned above, I had to download and modify the source code of the XSD2Code application and use it to generate classes for the ATML Tools. Since the XSD2Code source code is reasonably divided into several projects, I did not have hard time locating the file which needed to be edited. The problem was in a Xsd2Code.Library.Generator class, inside the method internal static Result<CodeNamespace> Process (GeneratorParams generatorParams) namely the Get XmlTypeMapping region. The source code did not search attribute group references recursively but just looked for the first level references. I fixed it with replacing the whole Get XmlTypeMapping region with the following code:

```
1  #region Get XmlTypeMapping
2
3  string rootFolder = "";
4  if (generatorParams.InputFilePath.IndexOf("\\") >
       1)
5      rootFolder = generatorParams.InputFilePath.
           Remove(generatorParams.InputFilePath.
           IndexOf("\\"));
6
7  XmlSchema rootSchema = XmlSchema.Read(XmlReader.
       Create(generatorParams.InputFilePath), new
       ValidationEventHandler(Validate));
8  var schemaSet = new List<XmlSchemaExternal>();
9
10 ExtractIncludes(rootFolder, rootSchema, ref
       schemaSet);
11 var schemas = new XmlSchemas { rootSchema };
12 schemaSet.ForEach(schemaExternal => schemas.Add(
       GetSchemaFromFile(rootFolder, schemaExternal.
       SchemaLocation)));
13
```

```
14 // Find and replace all nested attributeGroupRefs
      with their respective attributeGroups
15 // Keep processing until there are no
      attributeGroupRefs left that are more than one
      level deep
16 foreach (XmlSchema schema in schemas)
17 {
18     bool replacementOccurred = true;
19     while (replacementOccurred)
20     {
21         replacementOccurred = false;
22         foreach (XmlSchemaAttributeGroup group in
              schema.Items.OfType<
              XmlSchemaAttributeGroup>())
23         {
24             // Get the list of all the
                 attributeGroupRefs in this
                 attributeGroup
25             List<XmlSchemaAttributeGroupRef>
                 groupRefs = group.Attributes.OfType
                 <XmlSchemaAttributeGroupRef>().
                 ToList<XmlSchemaAttributeGroupRef
                 >();
26             for (int i = groupRefs.Count - 1; i >=
                   0; i--)
27             {
28                 // Locate the referenced
                     attributeGroup
29                 // and add all of its attributes
                     to the outer attributeGroup
30                 XmlSchemaAttributeGroupRef
                     groupRef = groupRefs[i];
31                 XmlSchemaAttributeGroup refGroup =
                      schema.Items.OfType<
                     XmlSchemaAttributeGroup>().
                     First(x => x.Name == groupRef.
                     RefName.Name);
```

```
32                  foreach (XmlSchemaObject att in
                       refGroup.Attributes)
33                      group.Attributes.Add(att);
34                  // Remove the reference
35                  group.Attributes.Remove(groupRef);
36                  replacementOccurred = true;
37              }
38          }
39      }
40  }
41
42  schemas.Compile(null, true);
43
44  var importer = new XmlSchemaImporter(schemas);
45  var exporter = new XmlCodeExporter(ns);
46  var mappings = new List<XmlTypeMapping>();
47  foreach (XmlSchemaType schemaType in rootSchema.
       SchemaTypes.Values)
48      mappings.Add(importer.ImportSchemaType(
           schemaType.QualifiedName));
49  foreach (XmlSchemaElement schemaElement in
       rootSchema.Elements.Values)
50      mappings.Add(importer.ImportTypeMapping(
           schemaElement.QualifiedName));
51  mappings.ForEach(exporter.ExportTypeMapping);
52
53  #endregion
```

Whole XSD2Code with the fix is placed in the appendix on the CD.

4 ATML Tools modular architecture

The following chapter will discuss collecting and preparing common requirements for the ATML Tools software architecture and the Graphical User Interface (GUI).

4.1 Problem analysis

The ATML standard consists of 6 parts describing different sections necessary to implement into testing process. Since the standard is currently in the trial-use state, the patches and updates are released on regular basis. The basic requirement is that the ATML Tools software has to be able to support current version implemented in Honeywell company as well as all versions released in the future. Hence came out the need of highly modular software and also an appropriate GUI.

Software applications in aerospace industry often last for 20 years and longer, therefore also the ATML Tools has to count with the ability to be extended with new ATML editors, or other editors which are somehow related to the testing industry - for instance the editor of the signal maps, or "TSF Creator" - a custom editor of the IEEE 1641 Test Signal Framework (TSF) library.

Another requirement is that the customer shall be able to configure the ATML Tools to his specific needs - add or remove editors and plugins. To provide this ability, the application shall be able to work with plugins developed by Honeywell, or even by a third party company. Hence we need to decide what technology will be used to provide high modularity and design the system architecture as robust as possible. Let us define important definitions first:

Extensibility is "a mechanism for manipulating host application objects or extending host functionality, sometimes referred to as automation. Generally made available via an object model published as part of a host's SDK."[3]

Add-In (also called Add-On, Extension, Plug-In, Snap-In) is "a customization; an assembly loaded by a host. Also, something that automates a host application (such as a client). Something that

provides additional functionality to a host (service). An add-in is custom code, usually written by a third party, that is dynamically loaded and activated, by a host application, based upon some context (such as host startup or document loading). The add-in extends the host application via the host's public Application programming interface (API) (for example, an object model or managed class library) which was made available to the add-in via an Software Development Kit (SDK)."[3]

4.2 Possible solutions

There are several frameworks providing support for exetensibility on the market. Of course only solutions working with .NET 4 and above have been considered as useable. Two technologies have been considered to be operable for the ATML Tools purposes:

- Managed Add-In Framework[1] (MAF)
- Managed Extensibility Framework[2] (MEF)

Detailed description of both of these frameworks is beyond the scope of this bachelor thesis, so I will provide just a brief overview of both technologies. Note: Certain level of extensibility can be achieved using reflection. This approach is only possible in certain situations and can not be considered correct. Despite this, the decision matrix (Fig. 6) contains System.Reflection as one of possible solutions just to provide wider scope of possibilities.

Managed Add-In framework

The MAF is a part of the .NET, introduced in the version 3.5. It's types are placed in a System.AddIn namespace. It specializes itself to loading and unloading the add-ins, their sandboxing and isolation. MAF allows add-ins to run in their own application domain, so crashing of one of the add-ins does not have to mean the whole application crashes along. "At the center of the MAF-based application is the

1. MAFOverview-http://msdn.microsoft.com/en-us/library/bb384200(v=vs.100).aspx
2. MEFOverview-http://msdn.microsoft.com/en-us/library/dd460648(v=vs.100).aspx

pipeline. Communication between add-ins and the host application is strictly regulated by the pipeline. The pipeline is dynamically created by MAF through a set of loosely-coupled components which are used to version, marshal and transform the data as it passes back and forth between the host and each loaded add-in.

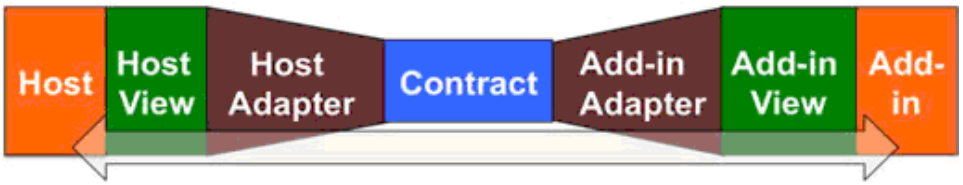


Figure 4.1: MAF Pipeline. Taken from [5]

Each section of the pipeline is contained in a separate assembly, loaded as necessary to manage the specific add-in. MAF discovers each component and then loads them on request using reflection.

For reliability, MAF allows optional isolation boundaries to be created between the add-in and host – everything to the left of the contract is loaded into the main (host) AppDomain and everything to the right loaded into a newly created AppDomain which has its own security permission set. The isolation can also be done through cross-process calls if true process-level isolation is necessary. Under the covers, the system uses traditional remoting calls to do the work of marshalling calls back and forth."[5]

Managed Extensibility Framework

The MEF project started in 2008 and has been included in the Microsoft .NET 4.0. Velvark, Novak and Granitz in the “Visual Studio 2010 and .NET 4 Six-in-One” book describe MEF as follows: “MEF as a technology is intended to enable greater reuse of applications and components by turning today’s statically compiled applications into dynamically composed ones."[4] The MEF types and attributes are declared in the System.ComponentModel.Composition namespace. It allows users to use extensions without any configuration required. Using MEF along with MAF is possible and the application can take advantages of both of them.

The application using MEF is composed out of parts. Part is an elementary building block of the MEF application. The parts specify their imports and exports. Imports can be objects or methods which you intend to use in your part. Exports are methods or objects that the part offers to be used by other parts. The parts are composed at runtime - the the imports are satisfied with the exports. The key feature for dynamic discovering available parts is the Catalog. There are several types of catalogs, most important are:

AssemblyCatalog - discovers parts in the given assembly

DirectoryCatalog - discovers parts in the specified directory

AggregateCatalog - can combine multiple catalogs

Obviously users can create their own implementation of catalogs (e.g. if there is a need to get the exports from a database, write your own DatabaseCatalog).

Catalogs are at runtime given as input to CompositionContainer and it performs composition from discovered parts.

Very important feature of the MEF is a lazy instantiation. "[...] during composition of a part, an import will trigger the instantiation of a part (or parts) that expose the necessary exports required for the original requested part. In the case of imported properties and fields, this means that their values are set before they are used. In many situations, it may happen that the properties are not used at all, or are used later after their original construction. If the instantiation of property values is expensive (takes a relatively long time, consumes many CPU cycles, holds a large amount of resources, and so on), it is a good approach to delay this instantiation — and prevent the recursive composition down the graph."[4] For this purpose MEF provides a new type of a generic container - Lazy<T>. The instances are now accessed through a Lazy<T>.Value property instead of direct access to an imported type. "The actual instantiation of the part is carried out when Value is fi rst queried, and subsequent queries use that cached instance."[4]

4.3 Conclusion

Like when choosing the classes generator, modular architecture technologies choice followed the QFD process. The following pros and cons table 4.3 and decision matrix 4.2 has been produced:

QFD		Requirement / Features (rule / approach)	System.Reflection		MAF (Managed Add-In Framework)		MEF (Managed Extensibility Framework)	
#	Customer Needs	Imprtnc Rank'g						
1	Performance as high as possible	30	3	90	5	150	7	210
2	Code maintainability as high as possible	30	1	30	5	150	7	210
3	Integration with used platform the smaller the better	30	10	300	7	210	7	210
4	Usage simplicity as high as possible	15	3	45	3	45	7	105
5	Application extensibility as high as possible	30	5	150	5	150	10	300
		Total	615		705		1035	
		Relative Ranking	3		2		1	

Figure 4.2: Modular Architecture QFD

	System.Reflection	MAF	MEF
Pros	Mature technology integrated into .NET Framework	Plugins can run in their own app domain	Allows loading plugins at runtime
			Manages dependencies between parts
			Lazy loading
Cons	Poor performance	Suitable for more extensive applications	Still in development

Figure 4.3: Modular Architecture Technologies pros and cons

As the QFD matrix shows, we chose the MEF framework.

4.4 Solution

The application is based on Windows Forms technology. It consists of the main application window which contains the main menu with items to create new or open ATML Files, options and with items for each available ATML editor. Therefore the AtmlToolsMainForm class serves as the main application form opened after the application start and it also serves as the host part of the MEF classes. Hence as the first method in constructor, the Compose method is called. The Compose method's content is following:

```
1  private static string EDITORS_PATH = Path.
       GetDirectoryName(Application.ExecutablePath) +
       "\\Libraries\\Editors\\";
2
3
4  /// <summary>
5  /// Contains imported instances of IAtmlEditor and
       theirs metadata.
6  /// </summary>
7  [ImportMany]
8  public IEnumerable<Lazy<IAtmlEditor,
       IAtmlEditorData>> Editors
9  {
10 get;
11 set;
12 }
13
14
15 public void Compose()
16 {
17       AggregateCatalog catalog = new
           AggregateCatalog();
18       catalog.Catalogs.Add(new AssemblyCatalog(
           typeof(Program).Assembly));
19       catalog.Catalogs.Add(new DirectoryCatalog(
           EDITORS_PATH));
```

```
20          container = new CompositionContainer(catalog
                );
21          try
22          {
23                  container.ComposeParts(this);
24          }
25          catch (CompositionException ex)
26          {
27              MessageBox.Show(ex.Message, "Composition
                    Error!");
28              throw;
29          }
30  }
```

The AggregateCatalog is used to combine two catalogs - an AssemblyCatalog locating the parts in the current assembly and the DirectoryCatalog, which is discovering the parts in assemblies located in the directory specified by a relative path. After the composition is completed, the main menu is created so that every found editor plug-in has it's own menu item and is shown in the project wizard. Whenever an assembly containing a plug-in is placed in the specific folder and the application is restarted, the new plug-in is ready to use.

5 GUI design

One of the main objectives when designing the application is to create an appropriate graphical user interface. In this chapter I will give an overview of requirements and best practises of GUI design and their implementation when designing the "ATML Tools".

5.1 Problem analysis

The goal is to design GUI so that it will be:

- useful
- effective
- as simple as possible
- following the CRAP [7] (contrast, repetition, alignment, proximity) principle

We have identified 10 types of users who will interact with ATML Tools:

1. Administrator: has access to all ATML tools components and is able to change access rights and ATML tools components configuration.
2. Product Design Engineer: creates and maintains UUT description documentation.
3. Product Manufacturer: creates UUT instance file for each instance of UUT.
4. Test HW design Engineer: creates and maintains Test HW description data (Instruments, Test Adapters, Test Stations) and read UUT and Test SW data.
5. Test SW design Engineer: creates and maintains Test SW description data (Test Description, Signal libraries) and reads the UUT and Test HW data.

6. Test Configuration Manager: creates and maintains Test configuration data and read all test system description data.
7. Test HW manufacturer: creates and maintains Test HW instance data (Instruments, Test Adapters, Test Stations).
8. Test System Domain Lead: inspects and analyzes all Test system description data.
9. ISC technician/ Maintenance: updates UUT instance file and analyzes archived reports.
10. AME / VE ISC Support: performs statistic analysis and data evaluation of archived reports and read test system environment data

5.2 Solution

Since 7 out of 10 identified users are users on engineering positions who are generally familiar with advanced engineering or development tools such as Microsoft Visual Studio or NI TestStand[1], we decided to design the ATML Tools so that it would look and behave similarly.

5.2.1 Main window

The first two goals was to make the user rearrange various editor windows and other tools such as a Project Explorer or a Properties window across the screen and to allow running application on multiple monitors, as the engineers often work with two and more screens simultaneously. To deliver these features, we used the DockPanel Suite.[2] The DockPanel Suite project declares itself as inspired by Microsoft Visual Studio and I have to confirm this statement. The DockPanel Suite is very easy to use - with minimum source code and configuration needed delivers full Visual Studio-like docking experience. Placement of singular windows is by default set as it is shown in figure 5.1.

1. http://sine.ni.com/np/app/main/p/docid/nav-94/
2. http://dockpanelsuite.com/

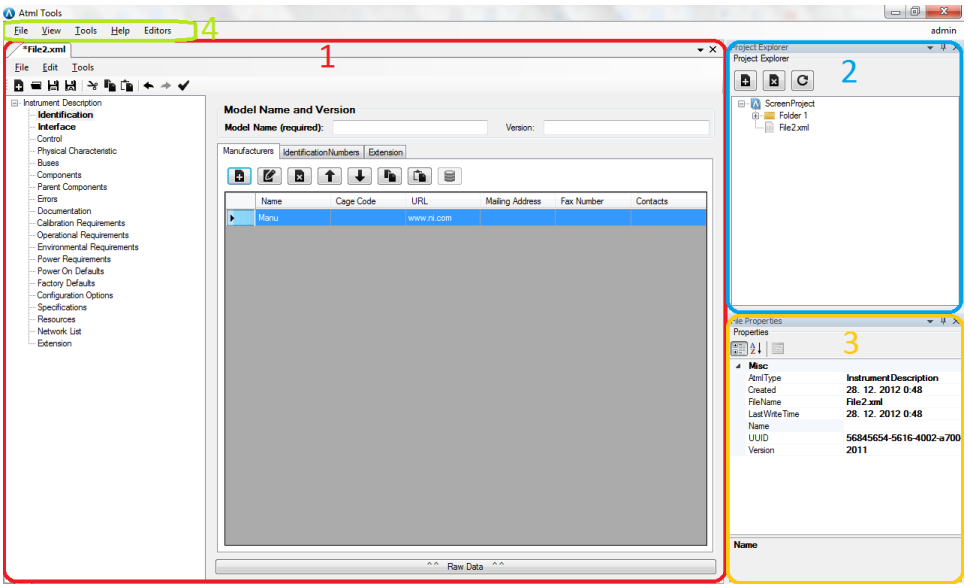


Figure 5.1: ATML Tools application window: 1 - Editor window, 2 - Project Explorer, 3 - File Properties window, 4 - Main menu

User can undock any window and move it to another desktop, or dock it to any part of the application window.

5.2.2 Editor window

Very important role plays the interface of the editors. Editor window is composed of a `TreeView` control placed on the left and a `Panel` component on the right, which acts like a parent container for components for editing singular sections of the ATML object. Treeview items for each ATML object are stored in the configuration file and are loaded in the constructor.

The `TreeView` object's `AfterSelect` event opens component specified in the configuration panel in the Panel on the right, where user can edit properties of the ATML object.

Editor also contains a toolbar with standard operations: *N*ew File, Open File, Save File, Save All, Cut, Copy, Paste, Undo, Redo and a *V*erify button.

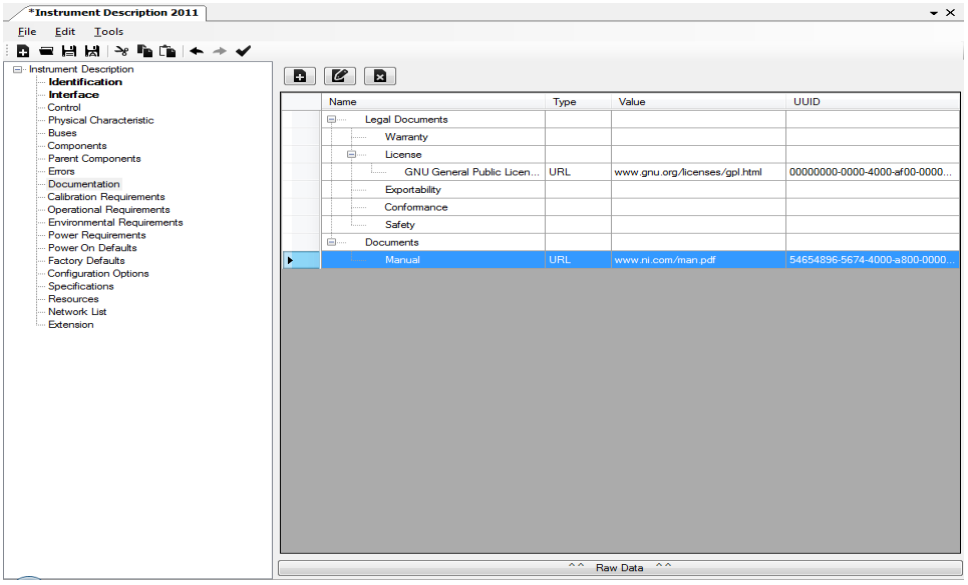


Figure 5.2: Editor window with `TreeView` and *Documentation* component

To unify icons look, the Default Icon[3] icon set has been used.

3. `http://www.defaulticon.com`

6 Conclusion

The goal of the thesis was to find out possible solutions of automatic generating source code from the XML schema data structures and provide useable C# classes generated from the IEEE 1671 standard. Three of possible solutions have been compared. It was decided to use the XSD2Code tool, however I had to fix the bug in the XSD2Code. The functional has been solution is provided and used in the ATML Tools software.

Another of the main goals was to find the possibilities to make the application extensible and describe a high-level architecture and solution. I compared two frameworks that provide wanted functionality and are included in the .NET 4 framework - MEF and MAF and then described the solution where we used the MEF framework.

The last chapter was about designing the GUI of the application. Since the whole application can not be disclosed, the screenshots of the final design are in the appendix.

The ATLM Tools software was successfully released on 5th December 2012. According to a plan it will be incorporated in the Honeywell CTS Core and it will be used across the company to implement new generation of automatic tests. The software will significantly contribute to rapid test development and the test results exchange between Honeywell sites across the globe.

6.1 Further improvements

The future versions of the ATML Tools will aim to the user comfort. Several enhancements are planned in the editors - the Instrument-Description's Networking section will have fully graphical interface. The ATML Tools will be extended with custom editors - "TSF Creator" and "Signal map editor" for a complex testing experience. Through using the MEF technology, it is enabled to accept third party editors. However there is still space to improve the genericity and make the application core more lightweight with less dependencies.

Bibliography

[1] Atml the standard for interfacing test system components using xml, 2011. This is an electronic document. Date of publication: August 3, 2011. Date retrieved: December 5, 2012. Date last modified: [Date unavailable]. URL: http://www.ni.com/white-paper/3893/en#toc7.

[2] Ieee standard for automatic test markup language (atml) for exchanging automatic test equipment and test information via xml. *IEEE Std 1671-2010 (Revision of IEEE Std 1671-2006)*, pages 1 –388, 20 2011.

[3] J. Gudenkauf and J. Kaplan. Clr inside out: .net application extensibility, 2007. This is an electronic document. Date of publication: February 1, 2007. Date retrieved: December 13, 2012. Date last modified: [Date unavailable]. URL: http://msdn.microsoft.com/en-us/magazine/cc163476.aspx.

[4] I. Novák, A. Velvart, A. Granicz, G. Balássy, A. Hajdrik, M. Sellers, G. Hillar, Á. Molnar, and J. Kanjilal. *Visual Studio 2010 and .NET 4 Six-in-One*. Programmer to programmer. Wiley, 2010.

[5] M. Smith. Creating extensible applications with maf, Dec. 2008. This is an electronic document. Date of publication: December 3, 2008. Date retrieved: November 21, 2012. Date last modified: [Date unavailable]. URL: http://www.developerfusion.com/article/9396/creating-extensible-applications-with-maf/.

[6] R. Taylor. Software tools: A key component in the successful implementation of the atml standards. In *AUTOTESTCON, 2011 IEEE*, pages 124 –129, sept. 2011.

[7] R. Williams. *The Non-Designer's Design Book: Design and Typographic Principles for the Visual Novice*. Non Designer's Design Book. Peach Pit, 2008.

List of Figures

3.1 XSD2Code GUI, taken from the XSD2CODE application *14*
3.2 Automatic code generation QFD *15*
3.3 Automatic code generation tools pros and cons *15*
4.1 MAF Pipeline. Taken from [5] *23*
4.2 Modular Architecture QFD *25*
4.3 Modular Architecture Technologies pros and cons *25*
5.1 ATML Tools application window: 1 - Editor window, 2 - Project Explorer, 3 - File Properties window, 4 - Main menu *30*
5.2 Editor window with `TreeView` and *Documentation* component *31*
A.1 New File Dialog *35*
A.2 InstrumentDescription Editor - Control - Driver - Tools *36*
A.3 InstrumentDescription editor - Control - Driver - Operating system *37*

A ATML Tools screen shots

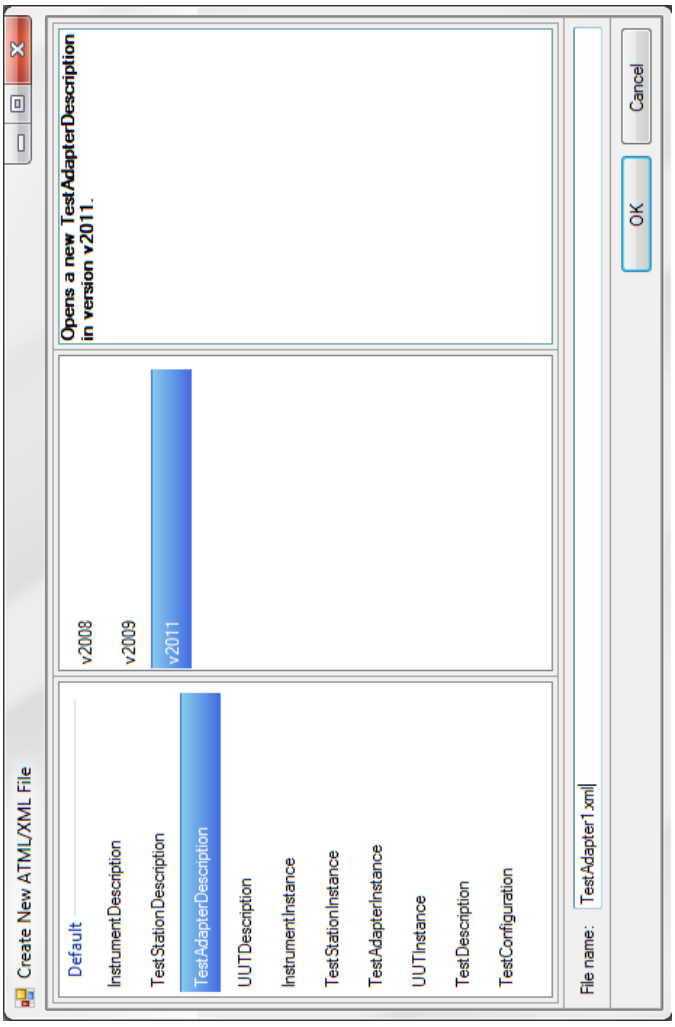


Figure A.1: New File Dialog

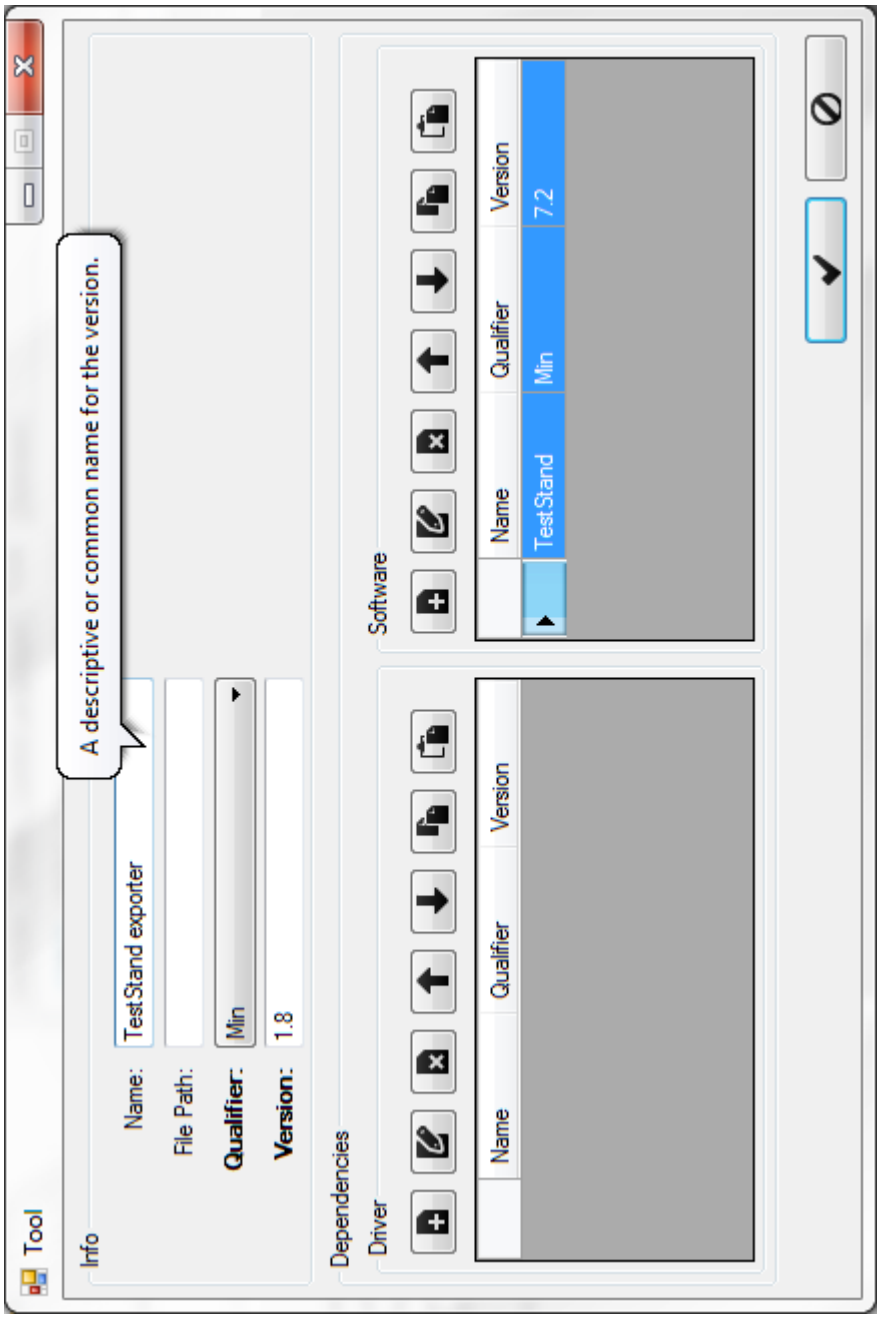


Figure A.2: InstrumentDescription Editor - Control - Driver - Tools

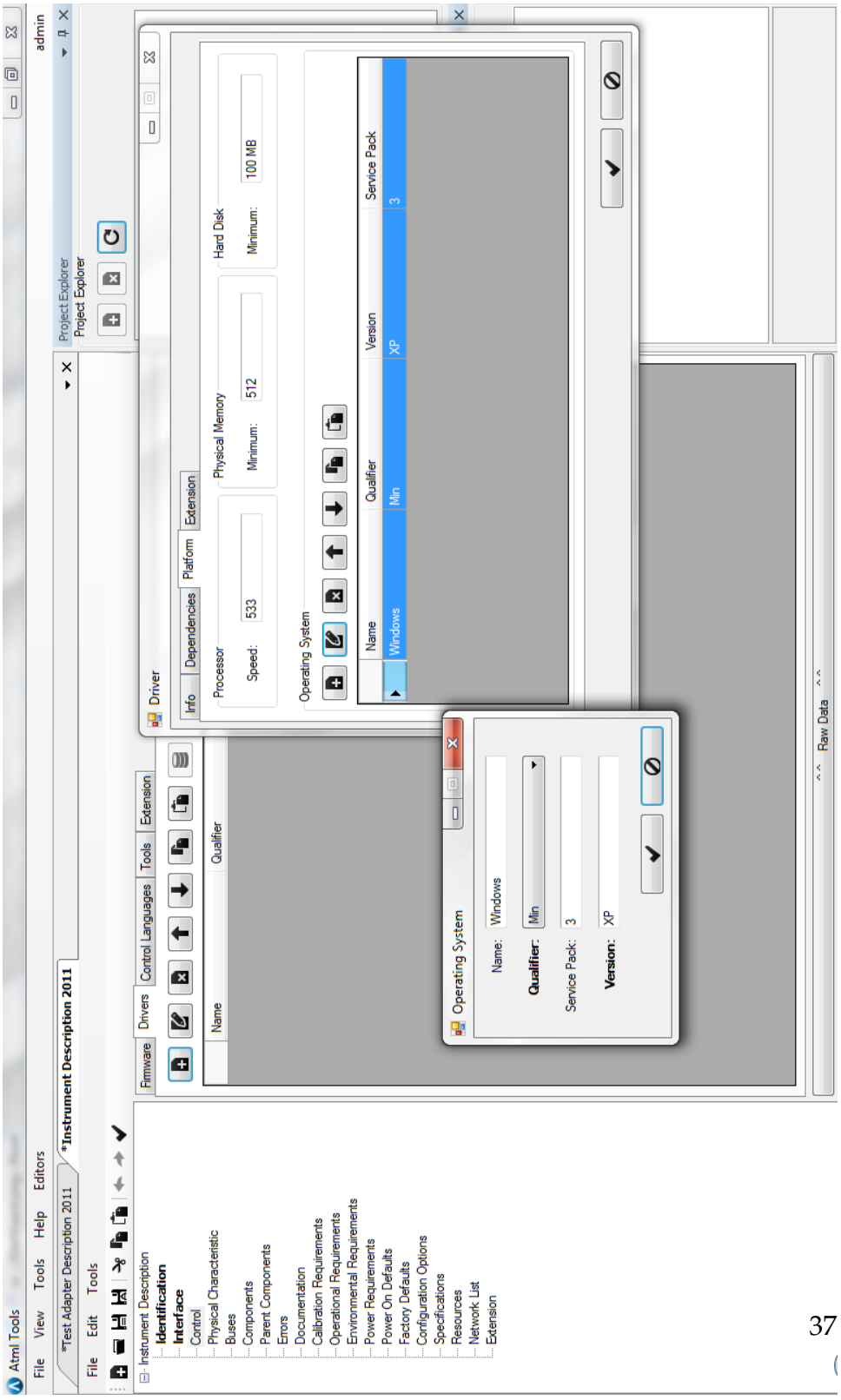


Figure A.3: InstrumentDescription editor - Control - Driver - Operating system

B CD content

The attached CD contains:

- XSD2Code source code
- Examples generated using Xsd.exe, LINQ to XSD and XSD2Code
- Screen shots of the ATML Tools GUI