LABCAR-AUTOMATION 4.2.2

ATCL (Automotive Test Class Library)
Getting Started for Test Case Developers

Contents

1 Introduction

This “Getting Started” shows the test case developer, how to create a test case for LABCAR-AUTOMATION V4.0 and higher using the Test Design Connector .NET (ATCL). The test design language is chosen to be C#, but the examples may easily be transferred to any other Microsoft .NET compliant programming language like VB.NET or Python.NET.

The tutorial shows in detail how to

- set up a new test case,
- call functionality of the test bench using ports and signatures
- generate necessary interface description to publish the test case in a Test Release Library
- use parameters for external parameterization in the Test Manager
- handle logging
- manage test bench configuration and initialization.

Please have fun with your first steps in LABCAR-AUTOMATION with .NET.

Your LABCAR Product Team

1.1 Preliminaries

The tutorial assumes that you have basic knowledge about the principles and terms used in LABCAR-AUTOMATION V2, V3 or V4. The Product Information presentation and the user manual will help. You also should have of rudimentary programming skills in C# or another object-oriented language.

Preparations:

- Installation of LABCAR-AUTOMATION V4.x with LABCAR-TBCNET or LABCAR-AUTOMATION Core V4.0 or higher. The LABCAR-AUTOMATION Core is part of each LABCAR-AUTOMATION package installation like Standard, Professional or Embeddable.
- Installation of Microsoft Visual Studio 2005 or later (or any development environment which allows creating C# code). This document anyway makes use of terms used in Microsoft Visual Studio.
- Installation of LABCAR-OPERATOR V4.0.x or higher and (optionally) INCA V5.4.1. or higher
- A compatible LABCAR-RTPC V3.0.x to execute the examples.

1.2 Installation Paths

All installation paths in this document refer to a **Windows 7** (64bit) system environment. The standard directories in this environment are:

Program Files:

```
C:\Program Files (x86)\ETAS\LABCAR-AUTOMATION 4.x
```

Configuration Files:

```
C:\ProgramData\ETAS\LABCAR-AUTOMATION 4.x
```

Examples and Default Data:

```
C:\Users\Public\Documents\ETAS\LABCAR-AUTOMATION 4.x
```

When working in a **Windows XP** environment, these directories are located at:

Program Files:

```
C:\Program Files\ETAS\LABCAR-AUTOMATION 4.x
```

Configuration Files:

```
C:\Documents and Settings\All Users\Application Data\
ETAS\LABCAR-AUTOMATION 4.x
```

Examples and Default Data:

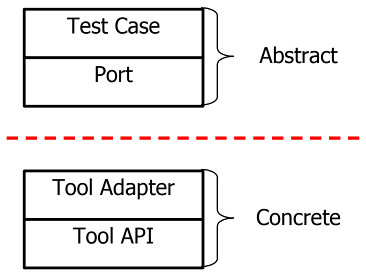
```
C:\Documents and Settings\All Users\Documents\
ETAS\LABCAR-AUTOMATION 4.x
```

2 ATCL Overview

2.1 What is the Automotive Testing Class Library (ATCL)?

The ATCL (Automotive Testing Class Library) is a framework which allows writing test bench independent test cases for usage in LABCAR-AUTOMATION. It is shipped with the product add-on LABCAR-TDCNET (Test Design Connector .NET) and can be found in the LABCAR-AUTOMATION installation folder ("/TestTools/bin/Etas.Eas.Atcl.Interfaces.dll").

The ATCL provides a large set of functions to access a test bench without using tool specific and proprietary methods. To keep the test case independent from any tool used, the ATCL provides a list of so-called ports, which group specific functionality (called "signatures") in a reasonable and native manner.



During the execution the LABCAR-AUTOMATION Test Bench Connectors take care of interpreting these ports and signatures and maps them into tool and APIs calls (without the tester having to deal with this personally).

2.2 Ports and Signatures

The following ports (with their signatures) are currently defined:

Table 1:

Port	Description	Name of port interface in the ATCL
Model Access	Used to access Test Bench simulation and modeling tools and any kind of simulation engines (such as LABCAR-OPERATOR or dSPACE ControlDesk). Functionality is available for setting and reading model values (measure elements and parameters), for data logging and signal generation and so on.	`IPortMA`
ECU Access Measurement	Used to access ECU internal data (measure elements, calibration data) and to flash an ECU	`IPortEAM`
ECU Access Calibration		`IPortEAC`
ECU Access Flash		`IPortEAF`
Sync DL	Working like a mixer to synchronize the data logging between different ports (e.g. to get synchronous data for INCA and LABCAR-OPERATOR	`IPortSyncDL`
File Access Composite	Ports providing and abstract methods to access either tabular (e.g. *.ini-or *.csv like) data or composite	`IPortFACD`

Data	(e.g. hierarchical data like *.xml-files) to read data into test cases without directly opening files from a test case.	
File Access Tabular Data		`IPortFATD`
Fault Simulation	Provides functionality to control fault simulation scenarios	`IPortFS`
J1699	Provides functionality to execute OBD compliance tests according to the J1699 specification	`IPortJ1699`
Diagnostics	Provides functionality to access the ECUs diagnostic (fault code) memory.	`IPortDiag`

For a detailed list of functions (“signatures”) available for each port, please refer to the “ATCL reference Manual” shipped electronically with LABCAR-AUTOMATION.

2.3 Managers

Next to the described ports to access test bench functionality, the ATCL provides a number of so-called “managers” dealing with test case internal data such as verdicts, parameters, reports and metadata.

In the test case, these managers are accessed using the ATCL “Factory” class. To make use of functions of the managers, you have to get a handle to them using the respective “Get…Manager” function of the Factory like

```
… Factory.GetMetaDataManager() …
… Factory.GetParameterManager() …
… Factory.GetVerdictMAnager() …
```

The use of this factory can be seen in more detail in the subsequent chapters.

3 Writing a simple test case

An ATCL test case which shall be executed by the LABCAR-AUTOMATION usually is a .NET executable. If a test case is used in a test project (*.lcaprj) and happens to be started by the test handler, it connects itself to the LABCAR-AUTOMATION and the needed tools depending the signatures used and the list of available tools described in a test bench configuration.

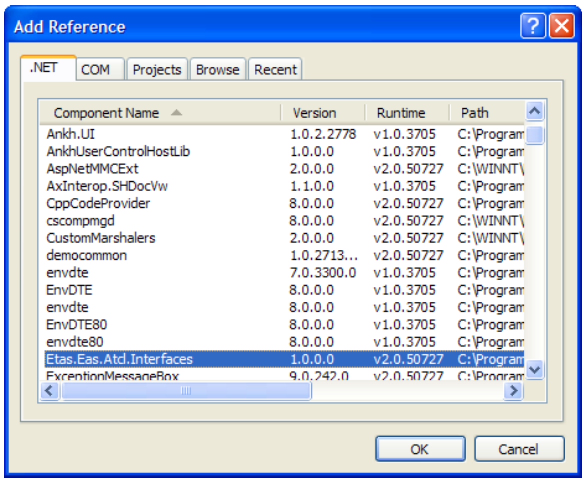
3.1 Setting up a new Visual Studio solution

You can write your ATCL test case in any language which supports a .NET connector and in any development environment you like. In this tutorial we assume that you are using Visual Studio 2005 and C#.

For a new test case you create a new "Console Application" using the Visual Studio's Project wizard. The wizard automatically creates a class and a main method for you.

```
namespace TC_Demo
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

In order to get access to the ATCL framework, you have to add a reference to the `Etas.Eas.Atcl.Interfaces.dll` to your project. In Visual Studio 2005 this DLL is automatically integrated by the LABCAR-AUTOMATION installation and can be found in the "Add reference" dialog.



Component Name	Version	Runtime	Path
Ankh.UI	1.0.2.2778	v1.0.3705	C:\Program
AnkhUserControlHostLib	1.0.0.0	v1.0.3705	C:\Program
AspNetMMCExt	2.0.0.0	v2.0.50727	C:\WINNT\
AxInterop.SHDocVw	1.1.0.0	v1.0.3705	C:\Program
CppCodeProvider	8.0.0.0	v2.0.50727	C:\Program
cscompmgd	8.0.0.0	v2.0.50727	C:\WINNT\
CustomMarshalers	2.0.0.0	v2.0.50727	C:\WINNT\
democommon	1.0.2713...	v2.0.50727	C:\Program
envdte	7.0.3300.0	v1.0.3705	C:\Program
EnvDTE	8.0.0.0	v1.0.3705	C:\Program
envdte	8.0.0.0	v1.0.3705	C:\Program
EnvDTE80	8.0.0.0	v1.0.3705	C:\Program
envdte80	8.0.0.0	v1.0.3705	C:\Program
Etas.Eas.Atcl.Interfaces	1.0.0.0	v2.0.50727	C:\Program
ExceptionMessageBox	9.0.242.0	v2.0.50727	C:\Program

Additionally the DLL can be found in the "PublicAssemblies" directory in the LABCAR-AUTOMATION installation directory.

In case you make use of any other DLL, you have to reference this also in the project.

Please keep in mind, that referencing .NET DLLs or COM components directly from the test case induces test bench dependent behaviour into the test case.

Once you added a reference to the ATCL interfaces you have to derive your class from the "Etas.Eas.Atcl.TestCase" base class. This is a class defined in the ATCL framework from which all test cases have to be derived. This derivation transforms a standard executable into an abstract test case ready to be used in the LABCAR-AUTOMATION tool chain. If we try to compile our test case now, we will get an error message since we did not implement the test case class's constructor yet.

3.1.1 Adding a Constructor

The `TestCase` base class does not have a default constructor (as known from other .NET classes) but a specific constructor taking the test case name as first argument. So we have to add a constructor which supplies the test case name to the base class. Additionally we rename our test case class and call it "MyFirstTestCase"

Now, the code should look like:

```
namespace TC_Demo
{
    public class MyFirstTestCase : Etas.Eas.Atcl.Interfaces.TestCase
    {
        public MyFirstTestCase()
   : base("MyFirstTestCase")
        {
        }

        static void Main(string[] args)
        {
        }
    }
}
```

3.1.2 Connecting to LABCAR-AUTOMATION

Now we can compile and execute our test case but there is still no connection to the LABCAR-AUTOMATION. This connection is established when we create an instance of our test case class.

```
static void Main(string[] args)
{
    MyFirstTestCase tc = new MyFirstTestCase();
}
```

When we create an instance of our test case class and then run the Visual Studio Project, we'll see the console for the test case executable and after a few seconds we'll get an error message: "[e] - Could not locate the server. Retry in a few seconds!" This error message says that the test case tried to get connection to the LABCAR-AUTOMATION Engine but no connection could be established because the engine is not loaded. This message is correct! The engine is typically opened by the Test Handler Application, which has not been started yet.

3.1.3 New Files

Additional to displaying an error message something more has happened when we started the test case. If we have a look into the directory where the test case was executed from (usually that's the `bin\Debug` directory within our Visual Studio project) there are some more files.

These new files are:

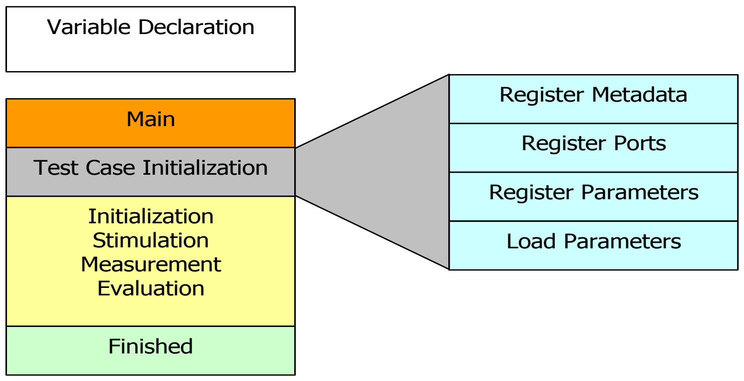
- a test parameter file (TPA)
- a test hierarchy description file (THD) and
- a test architecture description file (TAD).

These files are required by the LABCAR-AUTOMATION tool chain and are generated by the test case every time the test case is executed.

They contain information which ports are used by the test case, what parameters this test case requires for execution and some metadata information like test case version, test case developer etc. All information is generated at runtime. The ATCL framework records the corresponding framework calls and generates the appropriate files automatically.

4 Structure your Test Case

A test case should have a test case initialization section in which all preliminaries to the main test sequence are done (e.g. generation of test cases files, declaration of parameters and so forth). During this test case initialization section, you should avoid having exceptions being thrown and interrupting the procedure.



The underlying structure described in the following is not mandatory, yet it depicts the proceedings quite well.

4.1 Test Case Initialization

The Test Case Initialization section itself is more or less divided into four steps.

4.1.1 Register Metadata

At first we register the metadata for this test case. Metadata is any kind of prose, which is helpful to provide about a specific test case. Any kind of “key”/”value” pairs of strings (the first describing the metadata type and the second containing the data itself) can be used. Nevertheless, the minimum required metadata are a comment, a purpose and a version. These metadata items are generated automatically in case the test case developer does not explicitly specify them. Thus we should provide some valid information for them. As we can add all additional metadata we like, the example below also contains a company name as metadata:

```
private void RegisterMetaData()
{
     Factory.GetMetaDataManager().AddMetaData(
            new TCMetaData("Comment", "My first demo test case"));
          Factory.GetMetaDataManager().AddMetaData(
            new TCMetaData("Purpose", "Demonstrating how to write a
            simple test case"));
          Factory.GetMetaDataManager().AddMetaData(
            new TCMetaData("Version",
            GetType().Assembly.GetName().Version.ToString()));
          Factory.GetMetaDataManager().AddMetaData(
            new TCMetaData("Company", "ETAS GmbH"));
         }
```

The metadata is written to the THD file and we will see this metadata in the Test Manager when we add the test case to an LABCAR-AUTOMATION project.

4.1.2 Register Ports

The second step will be to register the ports we use in our test case. This step is important, since it tells LABCAR-AUTOMATION, which ports are used within the test case. During initialization of the test bench (opening the test bench tools and configuring them), LABCAR-AUTOMATION looks into this registration, to find out, which ports will be mapped to which tool.

You register a port "Model Access" and a port "ECU Access Measurement" using the following code

```
IPortMA  m_maPort = null;
IPortEAM m_eaPort = null;


...


private void RegisterPorts()
{
     m_maPort = Factory.GetPortMA("P_MA");
     m_eaPort = Factory.GetPortEAM("P_EAM");
}
```

The string passed to the respective "Get*" - method is the desired name of the port instance. It once again appears in the test bench configuration files (*.tbc), where it has to be mapped to specific tools. (see the "Test Bench Configuration" further below).

4.1.3 Register Parameters

The next step is the creation of a parameterization interface to provide the possibility of test adaptation by parameterization in the LABCAR-AUTOMATION Test Manager application.

Each test case may have an arbitrary structure of parameters, (which are specified in this section here) In the Test Release Library, the structure and specification of the parameters is stored in the "*.tpa"-file. By default a test case has no parameters and thus the automatically generated TPA file is always empty.

The parameterization consists of two steps. First we have to declare some variables as test case parameters, which are written to the TPA file and can be modified in the Test Parameter Manager. The second step is to read the parameterized value for that certain variable back into the test case (this is described in the following chapter)

Publishing variables as test case parameters is in some way similar to the registration of ports. First a local variable has to be declared. Then these variables have to be registered at the IParameterManager instance by calling the "Register()" method. Afterwards the parameters are written to the TPA file and can be displayed and modified with the Test Parameter Manager.

```
TypeSutFloat operandA = new TypeSutFloat(
     "Operand A", "operA", "First operand", 1.0, 0.0, 0.0, "");
TypeSutFloat operandB = new TypeSutFloat(
     "Operand B", "operB", "Second operand", 1.0, 0.0, 0.0, "");
```

```
...

private void RegisterParameters()
{
        Factory.GetParameterManager().CreateTpaFile();
        Factory.GetParameterManager().Register(operandA);
        Factory.GetParameterManager().Register(operandB);
        Factory.GetParameterManager().Save();
}
```

In the "RegisterParameters()" method the variables are registered at the ATCL and thus published as test case parameters. They will be written to the TPA file and can be configured by the TPM.

Keep in mind that this is only the first step. The values of the variables are still the default values defined in the constructor of the "TypeSUTFloat" . Even if the TPM would have set the parameters to some other values, this will have no effect in the test case until you explicitly read the parameterization for this test case.

4.1.4 Load Parameters

Before you use a variable which was published as test case parameter, you have to read the parameterized value for these variables. This is done by the "Parameterise()" method.

```
private void LoadParameters()
{
      operandA =
      Factory.GetParameterManager().Parameterise(operandA);
      operandB =
      Factory.GetParameterManager().Parameterise(operandB);


}
```

After the "LoadParameter()" method calls the value of the variables "operandA" and "operandB" have been changed from default values to the values supplied by the Test Parameter Manager. Consequently, if you don't call "Parameterise()" for a variable, you will always work with the default value of the variable.

IMPORTANT:
If you change or add parameters to the test case you need to re-run the test case one time to get the new parameters into the TPA file. Otherwise the TPA file will contain the old parameters.

4.2 Test Case Structure

After this test case initialization section we structure our test case into four different sections: Initialization, stimulation, measurement and evaluation section.

4.2.1 Initialization

In the initialization section we are preparing our test bench. This includes configuring all ports as well as configuring a data logger or e.g. set certain environment conditions. Typically the measurement is stopped during that initialization.

4.2.2 Stimulation

In the stimulation section we stimulate the ECU and/or the model to get the desired behavior. Stimulation can also be done during the measurement.

4.2.3 Measurement

In the measurement section we record values from the ECU and/or the model.

4.2.4 Evaluation

In the evaluation section, we compare e.g. the recorded values from the model with the recorded values from the ECU. We have to decide whether or not the recorded ECU values are OK.

4.3 Verdict Handling

A verdict has a built-in state machine. Verdicts can change from "none" to "pass", from "pass" to "fail", from "fail" to "inconc" (inconclusive) and from "inconc" to "error". Setting a verdict is done by calling the respective verdict function (e.g. "Pass()" or "Fail()"). If a verdict e.g. is "fail" you can never set this verdict back to "pass". Because of that, we suggest creating a new verdict for each internal section of a test case. This allows us to have some sections fail while other sections may pass. Verdict and sections are closely coupled to report design and thus have a look at Chapter 5 for more details on that.

4.4 Configuring your test bench

In order to get our test bench running, you need to configure it. Configuring the test bench is a three step process:

1. Mapping ports to tools

   In the test bench configuration files (*.tbc, edited with the TBC Editor) all required port instances have to be mapped to a tool (by specifying the respective tool adapter). For each tool, a tool configuration is provided, listing available tool specific project data.

2. Mapping of tool configurations

   In the Tool Configuration Files (*.tcf, edited with the TCF Editor), each tool specific project (e.g. LABCAR-OPERATOR projects, INCA databases) are abstracted using sets of IDs. There is typically more than one single ID identifying a tool configuration to allow a configuration dependence both from the test case and from an UuT. For each tool configuration item, one typically has a signal mapping (SUT Mapping) to keep the access to tool labels as abstract as possible.

3. SUT Mapping

   Each used signal or measure element label is mapped to a tool specific label in the SMF files (edited with the SUT Mapping File Editor). Mapping entries contains unit and value range specifications.

To completely configure the test bench, each port contains the same configuration signatures.

- Port_xy. Create()

  Opens the tool which is realizing the ports functions as specified in the TBC file.
- Port xy.ConfigureTool()

  Passes the first part of a set of tool configuration IDs to LABCAR-AUTOMATION. These IDs are defined for each UuT in the UuT Environment. Thus, this signature depicts the UuT dependence of a test bench configuration.
- Port xy.Configure()

  Passes the second part of a set of tcf IDs to LABCAR-AUTOMATION and thus completes the set of IDs. This signature is typically used in the test case itself to influence the test bench configuration from the test case.

With this last signature the tool configuration can be resolved and the respective tool is configured. For LABCAR-OPERATOR this means opening the *.lca-file specified in the tcf file, downloading the code to the LABCAR-RTPC and starting the target OS.

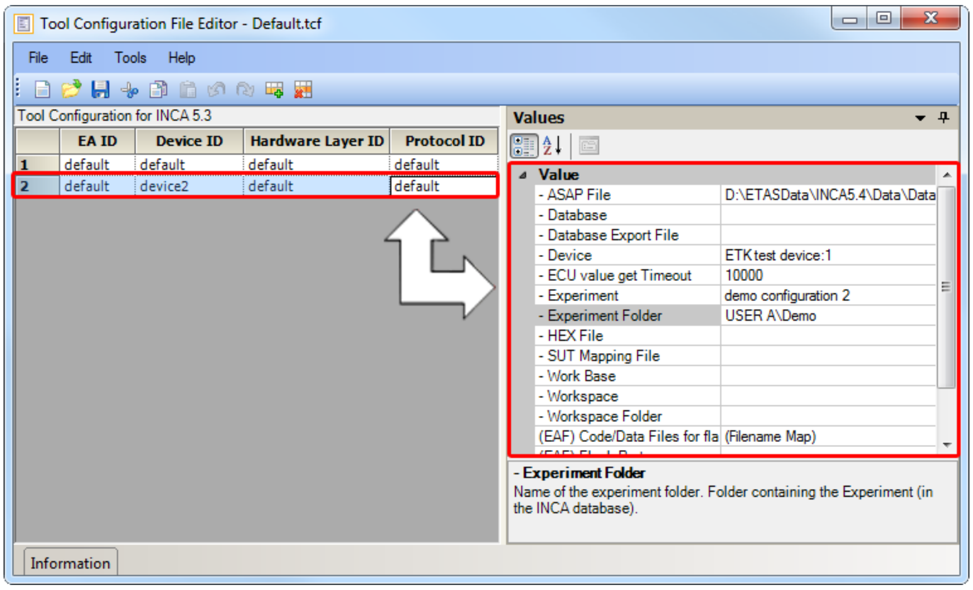
In most test projects, you will make use of the so-called test bench initialization ("TBInit"), which means, that the "Create()" and "ConfigureTool()" signatures of all required ports are called by the Test Handler tool before the first test case itself starts. Consequently, you do not need to code these explicitly in your test case.

In case you want to execute the test case directly from Visual Studio (e.g. for debugging purposes), you have to call the "Create()" and "ConfigureTool()" signatures yourself within the test case code. The test case examples show several possibilities, how to handle this within the initialization.

The signatures "ConfigureTool()" and "Configure()" are port specific and have port specific arguments. This means, for each port it has been separately specified, how many IDs are required to identify a configuration, and in which way they are separated between "Configure()" and "ConfigureTool()". The configuration for the port "Model Access" is mapped using three IDs which are all passed by the "ConfigureTool()" signature, while the tool configuration for the port "ECU Access" is mapped using four IDs of which only one is passed to LABCAR-AUTOMATION with the "ConfigureTool()" signature, while the other three are passed from the test case using the "Configure()" signature.

An example for the EAM port shall help to illustrate this: The EAM port has four configuration keys "EA_ID", "DeviceID", "HWLayerID" and "ProtocolID". The combination of the four keys selects exactly one entry from the TCF file.

In the screenshot below there are two different EAM port configurations. The first configuration in the example is identified by the keys "default", "default", "default" and "default" while the second configuration is identified by the keys "default", "device2", "default" and "default".



Tool Configuration for INCA 5.3

	EA ID	Device ID	Hardware Layer ID	Protocol ID
1	default	default	default	default
2	default	device2	default	default

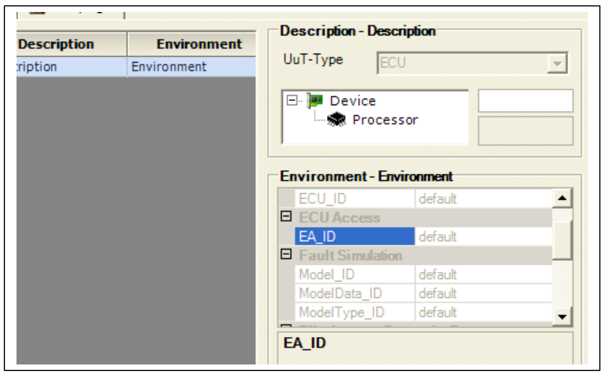
Value

- ASAP File	D:\ETASData\INCA5.4\Data\Data
- Database	
- Database Export File	
- Device	ETK test device:1
- ECU value get Timeout	10000
- Experiment	demo configuration 2
- Experiment Folder	USER A\Demo
- HEX File	
- SUT Mapping File	
- Work Base	
- Workspace	
- Workspace Folder	
(EAF) Code/Data Files for fla	(Filename Map)

- Experiment Folder
Name of the experiment folder. Folder containing the Experiment (in the INCA database).

In the EAM port the "ConfigureTool()" signature takes only one argument which is the EA_ID configuration key. The "Configure()" signature takes three arguments which are "DeviceID", "Hardware Layer ID", and "ProtocolID". So once the test case called the "Configure()" signature, all required configuration keys are fixed and the configuration for the port instance can be loaded by the tool adapter.

The test case developer can now decide to either implement "ConfigureTool("default")" and "Configure ("device2","default","default")" or just call "Configure()" and let the Test Handler do the "ConfigureTool()". In case of a "TBInit" campaign the TestHandler calls "ConfigureTool()". The value for the "ConfigureTool()" call is defined in the UuT Environment of a UuT.



ECU_ID	default
ECU Access	
EA_ID	default
Fault Simulation	
Model_ID	default
ModelData_ID	default
ModelType_ID	default

EA_ID

The screenshot shows a part of the UuT Environment. For the port "ECU Access" there is only the EA_ID key, which is set to default. The values from the UuT environment are read by the Test Handler and passed to the "ConfigureTool()" signature of the corresponding ports. In case of the port "Model Access" for example the UuT Environment contains all three keys required to select a tool configuration. Thus the signature "Configure()" of the port "Model Access" does not have any arguments, because all required keys are already passed during the "ConfigureTool()" call.

In general we say: The keys from signature "ConfigureTool()" together with the keys from the signature "Configure()" select the configuration. The number of the "ConfigureTool()" and "Configure()" keys is port-dependent. There are ports with keys on both signatures like the "EAM" or "EAC" port, there are keys with exclusively "ConfigureTool()" keys like the "Model Access" port and there are ports with exclusively "Configure()" keys the "SyncDL" port.

ConfigureTool	Configure		
EA_ID	Device	HWLayer	Protocol

e.g. INCA Configuration

ConfigureTool			Configure
Model	Model Data	Model Type	NONE

e.g. LCO Configuration

ConfigureTool	Configure		
NONE	Key 1	Key 2	Key N

e.g. SyncDL Configuration

Additionally not all ports require a tool configuration file. Which tool configuration file (TCF) applies to a port is defined in the test bench configuration (TBC) file. The TBC file defines the list of available ports and maps these ports to tools respectively tool adapters. One tool may have zero or one TCF file. So if you are using many EAM port instances in your test case, they all work with the same TCF file.

Best Practice

If you do not have any special requirements on the "Configure()" keys you should use "default" as standard value. This makes it easier to create correct TCF entries for your test case.

Of course you have to ensure that all ports you are using in your test case are defined in the TBC file!

For more information about the configure signatures of certain ports, please have a look into the "ATCL Reference Manual".

4.4.1 Test Case Definition File

In order to add our new test case (functionality) to an LABCAR-AUTOMATION project, the additional files TPA (test parameter file), THD (test hierarchy description file), TAD (test architecture description file) and TCD are required next to the test case functionality itself. The TPA, THD and TAD files are generated by the test case automatically. Simply ensure that you have executed your test case at least one time before trying to add it to an LABCAR-AUTOMATION project.

The TCD file (test case definition) is still missing. It can be considered as "the test case". It defines which files belong to the test case. As said before, there have to be at least the TPA, THD and TAD file together with a test case itself which usually is an executable. The TCD file may define additional dependencies for the test case like Python scripts or .NET libraries.

The XML schema (lca_tcd_v3.0.xsd) for the TCD file can be found in the LABCAR-AUTOMATION installation directory in the ./TestTools/XMLSchema folder. The minimum TCD file for our simple test case will be:

```
<?xml version="1.0"?>
<file …="" TDL="StdExe" …="" >
  <TestCase Name="MyFirstTestCase">
    <EntryPoint>MyFirstTestCase.exe</EntryPoint>
  </TestCase>
  <MetaData>
    <TPA>MyFirstTestCase.tpa</TPA>
    <TAD>MyFirstTestCase.tad</TAD>
    <THD>MyFirstTestCase.thd</THD>
  </MetaData>
  <Checksums />
</file>
```

In the example XML you can see a minimal TDC file for our test case. You may recognize that the "Checksums" element is empty. The checksums for each file, belonging to that test case, are calculated when the test case is added to an LABCAR-AUTOMATION project. Anyway the "Checksums" element has to be present. The required files are listed in the "Metadata" element. Additional files needed by the test case are included by using a "Dependencies" element parallel to the "EntryPoint" element.

```
…
  <TestCase Name="MyFirstTestCase">
    <EntryPoint>MyFirstTestCase.exe</EntryPoint>
    <Dependencies>
      <Dependency>AdditionalFunctions.dll</Dependency>
    </Dependencies>
  </TestCase>
…
```

The TCD file also defines the Test Design Language (TDL). This attribute is used by the Test Handler to distinguish which test case executor has to be taken to execute that particular test case. In this case the executor for standard executables is selected. The "TDL" attribute has to match one executor defined in the TestCaseExecutorFactory.config file located in the TestTools/bin directory.

The LABCAR-AUTOMATION 4.0 installation supports:

- The "TTCN Executor" which is able to execute TTCN-3 test cases and is selected by the "TDL" attribute value "TTCN". The difference between the "TTCN Executor" and the "StandardExe Executor" is that the former provides some TTCN specific command line arguments like TTCN verbosity level etc.
- The "StandardExe Executor" which is able to execute "*.exe" files and is selected by the TDL attribute value "StdExe"
- The "Python Executor" which is able to execute Python scripts and is selected by the TDL attribute value "PythonNET"
- The "TestStand Executor" which is able to execute TestStand test sequences and is selected by the TDL attribute value "TestStand"

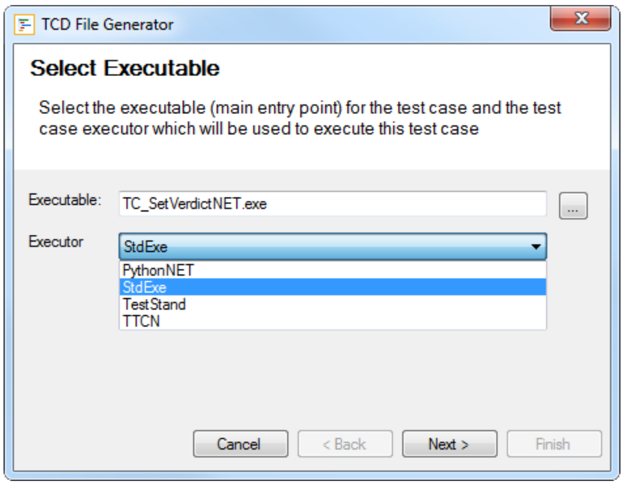
IMPORTANT
The TDL attribute value is case sensitive! If you have other test design languages with their corresponding executors you can select them by using their TDL value in the TCD file.

IMPORTANT
All dependencies, including the TPA, THD and TAD files have to be located within the test case directory. Referencing dependencies which are located outside the test case directory is not allowed nor are subdirectories beneath the test case directories.

4.4.2 Using the TCD File Generator

The TCD File Generator (TCDCMD.Gui.exe) comes with the LABCAR-TDCNET package and can be found in your Start menu under **Programs → ETAS → LABCAR-AUTOMATION → Test Design (ATCL) → TCD Generator**. This tool is a wizard which automatically creates the TCD file for you.

In the first dialog window you have to select the test case executable. This is the file which is considered to be "the test case". Additionally you have to select a test case executor which is used to execute the test case. For an ATCL C# .NET test case this will be the Standard Executor (StdExe).



Select Executable

Select the executable (main entry point) for the test case and the test case executor which will be used to execute this test case

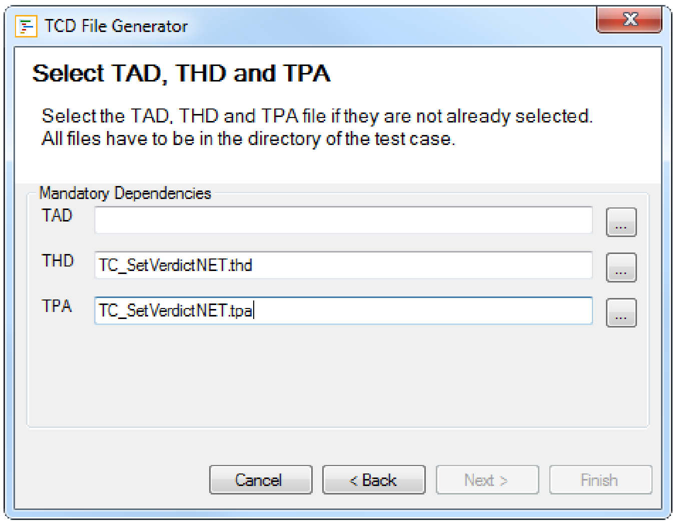
Executable: TC_SetVerdictNET.exe

Executor: StdExe

PythonNET
StdExe
TestStand
TTCN

If there is exactly one TPA, THD and TAD file besides the test case executable, these files are automatically taken as dependencies and if you do not need any other files for your test case, just click **Finish**.

If the wizard cannot detect the TAD, THD or TPA file you have to select them manually. To do so, use the browse button (...) to open a file selector window which lets you select the file. Remember that all files have to be within the test case directory.



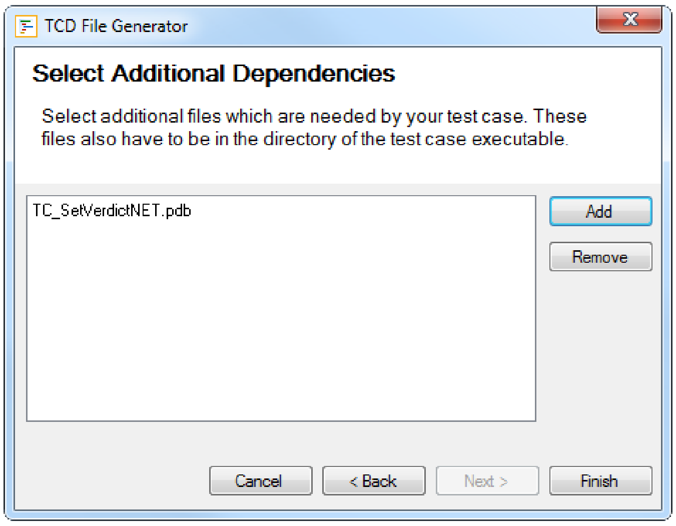
Select the TAD, THD and TPA file if they are not already selected.
All files have to be in the directory of the test case.

THD: TC_SetVerdictNET.thd

TPA: TC_SetVerdictNET.tpa

Once all required dependencies are defined, you can click either **Finish** to generate the TCD file or you may continue to the next window where you can select additional dependencies for your test case.

In the third window you can select additional dependencies by clicking the **Add** button. This opens a file browser dialog within the test case directory where you can select all kind of files. You must not select any file which is already a dependency of your test case (e.g. the TPA file) nor must you select any files in subdirectories or directories outside the test case directory.



Select additional files which are needed by your test case. These files also have to be in the directory of the test case executable.

TC_SetVerdictNET.pdb

The TCD file which is generated has the same name like the test case executable but the file extension is changed to ".tcd". The Test Manager's functionality browser cannot find your test case if you don't supply a TCD file! But you have to generate the TCD file only once for a test case. If you change your test case and then recompile it, you can just synchronize your LABCAR-AUTOMATION project what automatically updates the test case.

But you have to generate a new TCD file if you add (or remove) some additional resources like DLLs to the test case!

4.4.3 Using the Command Line Interpreter

Parallel to the TCDCMD.Gui.exe there is another application TCDCMD.exe which is command line tool generating the TCD file without showing a wizard. This can be used for instance as a post execution step in your Visual Studio development environment.

Usage

```
tcdcmd.exe <testlanguage> <entrypoint> <tpa> <thd> <tad> [<dependency1> [<more dependencies>]]
```

<testlanguage>	The language of the test case. Possible values are: "StdExe", "TTCN", "PythonNET" and "TestStand"
<entrypoint>	Start up file of the test case. Typically the executable.
<tpa>	Test Parameter File of the test case.
<thd>	Test hierarchy definition of the test case.
<tad>	Test architecture definition of the test case.
<dependency>	A additional file required by the test case to operate.

4.5 Setting up an LABCAR-AUTOMATION project

To actually execute a test case, the best way to go is to create a small LABCAR-AUTOMATION test project. The following steps are required to create a minimum test project for one test case and one UuT:

1. Use the UuT Editor to define an example UuT with an UuT Environment containing all the keys for the ports/tools. You might need to create a new UuT List.
2. Create a test project in the Test Manager tool and add your test case and the UuT to it.
3. Use the "Parameter" tab to assign any value to the two float parameters you created.
4. Create a test campaign containing the test case. You have to decide whether or not to use a "TBInit"- Campaign. This depends on our test case. If the test case does "Create()" and "ConfigureTool()" on its own, we use a non-TBInit campaign. But keep in mind that if the test case is a non-TBInit test case, we cannot change the TCF keys for that test case. It is also possible to create test cases which can handle both TBInit and non-TBInit campaigns as mentioned above.

5 Report Design

5.1 Sections and Verdicts

You might want to structure your report into different sections e.g. the sections mentioned in chapter 4. For each section you can create a new verdict which can be independent of the overall test case verdict. This allows you to have a section failed and following sections passed. Due to the verdict state machine this is not possible with the overall test case verdict. To get a section verdict, we have to create a new instance of a verdict.

```
Verdict sectionVerdict = new Verdict(VerdictCode.None);

try
{
     Reporting.SectionBegin(MethodInfo.GetCurrentMethod().Name);

     …

     sectionVerdict.Pass();
     Pass();//Overall test case verdict
}

catch (Exception ex)
{
     sectionVerdict.Error();
     throw ex;
}

finally
{
     Reporting.SectionFinished(sectionVerdict.ActualVerdictCode.ToString,
     sectionVerdict);
}
```

The code example may be applied to any method which shall appear as section in the test case report. The section name is read by the "MethodInfo.GetCurrentMethod().Name" method and returns the name of the current method.

5.2 Working with Tables

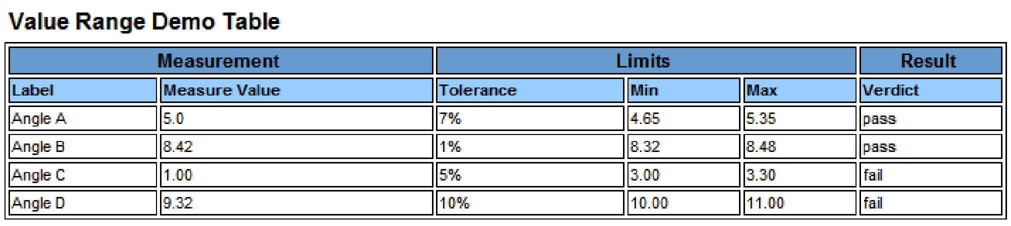
In order to display measured values in contrast to the allowed value ranges you might want to use tables in your report. Tables can be generated via the reporting manager. Tables are created "inside" the report service and are accessed and identified by their name. A created table is not added to the report automatically. This has to be done by the "AddTable" method, which adds the table with the specified name at the current location in the report.

```
string tableID = "table_ID";
Reporting.CreateTable(tableID, "Value Range Demo Table", 1, 0);
Reporting.SetTableHeadline(tableID, "Measurement",0 , 1);
Reporting.SetTableHeadline(tableID, "Limits", 2, 4);
Reporting.SetTableHeadline(tableID, "Result", 5, 5);

Reporting.SetTableColDescription(tableID,
     new string[] { "Label", "Measure Value", "Tolerance", "Min",
"Max", "Verdict" });

Reporting.SetTableData(tableID,
     new string[] { "Angle A", "5.0", "7%", "4.65", "5.35", "pass"
});
Reporting.SetTableData(tableID,
     new string[] { "Angle B", "8.42", "1%", "8.32", "8.48", "pass"
});
Reporting.SetTableData(tableID,
     new string[] { "Angle C", "1.00", "5%", "3.00", "3.30", "fail"
});
Reporting.SetTableData(tableID,
     new string[] { "Angle D", "9.32", "10%", "10.00", "11.00" ,
"fail"});
Reporting.AddTableToReport(tableID);
```

The resulting table will look like this:



Value Range Demo Table

Measurement		Limits			Result
Label	Measure Value	Tolerance	Min	Max	Verdict
Angle A	5.0	7%	4.65	5.35	pass
Angle B	8.42	1%	8.32	8.48	pass
Angle C	1.00	5%	3.00	3.30	fail
Angle D	9.32	10%	10.00	11.00	fail

5.3 Working with Plots

Working with plots is very similar to working with tables. You also have to use the reporting manager to create a new plot. The plot is also identified with a unique name but we additionally have a reference to the plot object within the report service. In order to add data to a plot you have to specify at least the X axis with and one or more Y axis.

If you add a line to the plot, you have to select the Y axis the data refers to. The plot itself is added to the report when you call the "AddPlot2Report" method and adds the plot to the current location in the report; similar to the table.

Typically the plot is used to display data recorded by a data logger.

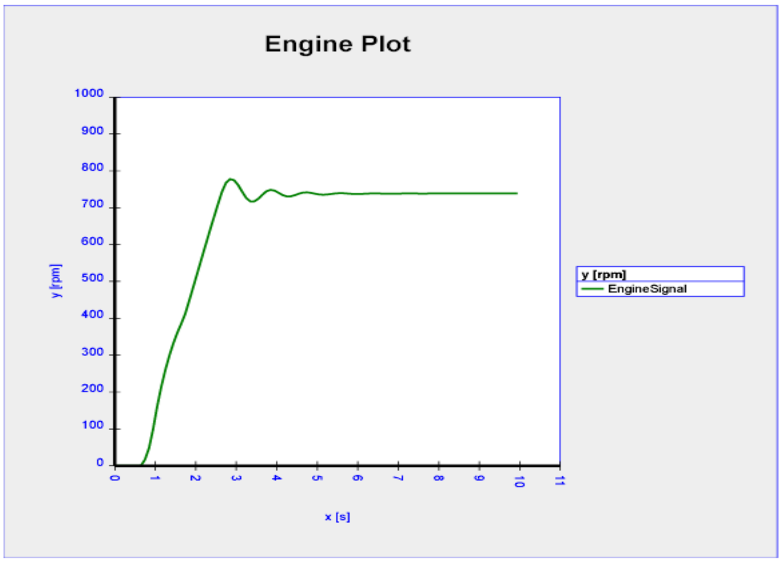
```
TypeSut1DFloatTable [] table = ... ;

IPlot plot = Reporting.CreatePlot ( "Plot", "Engine Plot", 1.0 );

plot.AddYAxis ( "y",0.0, 1000.0,"rpm",100.0);
plot.SetXAxis ( "x",0.0,11.0,"s",1.0);
plot.YAxisCollection[0].AddLine(
    "EngineSignal",
    table[0].ValueX,
    table[0].ValueY,
    0.0,
    0.0,
    plot.CreateLineFormat("green", LineWeight.Thin,
LineStyle.Stroke));

Reporting.AddPlot2Report (plot);
```

The data to be displayed was recorded by a data logger and retrieved via the "GetLoggedSignals(...)" method. The resulting plot will look like:



6 Using own software components

It is possible to integrate your own software components into the LABCAR-AUTOMATION and the ATCL. You can write your own port and tool adapters in order to integrate new tools. You can create your own managers and integrate them directly into the ATCL factory and you can use other DLLs within your test case.

6.1 ATCL Managers

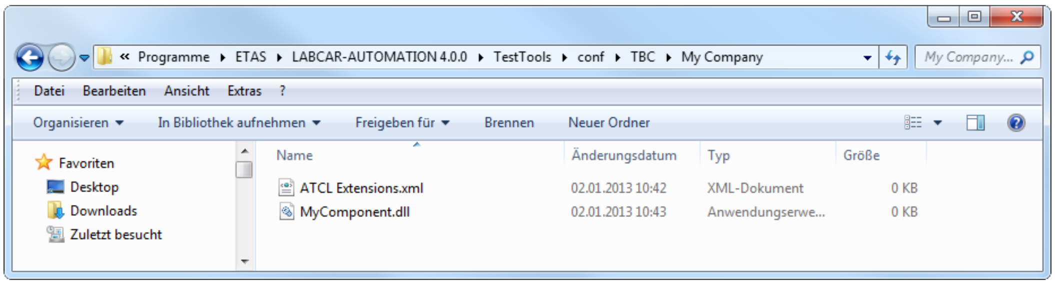
The ATCL factory by default provides managers for parameterization, reporting and metadata information. You can extend the list of available managers. Managers are anonymous software components which implement the "Etas.Eas.Atcl.Interfaces.Managers.Imanager" interface. You can retrieve an instance of a manager from the ATCL factory and then you have to cast that instance to the interface this managers implements.

For example you can use "Factory.GetReportManager()" to get an instance of the "IreportingManager" or you can use the general "Factory.GetManager("IReportingManager")" method to get the same instance. The difference is that you have to cast the reference to the "IreportingManager" interface.

Managers are used when you want to provide functionality which does not necessarily needs a tool adapter or a port. For example if you want to create a software component which wraps the generation of reporting tables. This functionality should be encapsulated into a separate assembly and accessed via the manager concept.

A custom manager can be inserted in the ATCL by adding a file ATCL Extension.xml somewhere beneath the <Install Path>/TestTools/conf directory. In this xml file you can declare your custom manager and afterwards you may access this manager via the method "Factory.GetManager("<unique manager name")".

An example of an ATCL extension can be found in the file TestTools/conf/TBC/ETAS/ATCL/ATCL Extensions.xml, which defines the FS port. The assembly which includes the required manager has to be located somewhere beneath the conf directory, too. We suggest keeping ATCL Extension.xml together with your DLL in a new subdirectory.


« Programme ▸ ETAS ▸ LABCAR-AUTOMATION 4.0.0 ▸ TestTools ▸ conf ▸ TBC ▸ My Company
My Company...
Datei Bearbeiten Ansicht Extras ?
Organisieren ▾ In Bibliothek aufnehmen ▾ Freigeben für ▾ Brennen Neuer Ordner
Favoriten
Desktop
Downloads
Zuletzt besucht

Name	Änderungsdatum	Typ	Größe
ATCL Extensions.xml	02.01.2013 10:42	XML-Dokument	0 KB
MyComponent.dll	02.01.2013 10:43	Anwendungserwe...	0 KB

Because managers are published by configuration, they are installation- and machine-depended. So you have to ensure that the machine which executes your test case has the required managers configured.

6.2 Ports and Tool Adapters

Whenever you want to access a new tool you should take your time and clarify, if the functionality which the tool provides is already available in an existing port. If so, you should write a new tool adapter for that tool and you are finished. Just map the existing port to your new tool adapter and use the ports signatures to control your new tool.

If there is no existing port you should think of creating a new port and tool adapter in order to gain access to your tool from an ATCL test case. Although there is some effort to create a new port and tool adapter, this should be the preferred way because this ensures that your test cases remain tool independent.

New ports are defined exactly like managers in the file ATCL Extension.xml. The only difference is that ports are queried from the ATCL factory via the "GetPort (<identifier>, <instance name>)" method. "identifier" is the id of the object within ATCL Extension.xml.

```
<object id="IPortFS"
type="Etas.Eas.Atcl.Core.Port.FailureSimulation.FailureSimulationPor
t, Etas.Eas.Atcl.Core" singleton="false" />
```

Every port requires an additional XML file which defines the available signatures, in which states these signatures are allowed to be called and some more information. The XML file fully describes the new port. Because creating this XML is time consuming and error-prone, we (ETAS) have an XML generator which creates all required files out of the port's interface. This generator is NOT part of the product installation, but we will create the files for you for free, if you provide the port's C# interface implementation.

Creating ports is not the daily business of testers and test case designers, so we restrict ourselves to a quick description here. Please feel free to contact ETAS to clarify on consulting and engineering services to support your use case here.

6.3 Using other DLLs

Additionally to the manager and the port concept you may also reference a needed assembly directly by adding the reference to the Visual Studio project. If your test case requires additional resource assemblies, you have to specify them in the TCD file. Otherwise the referenced assemblies are not considered as part of your test case and thus are not copied into the LABCAR-AUTOMATION project, when the test case is added. The benefit of referencing the assembly directly is that you can execute that test case on any LABCAR-AUTOMATION test bench.

7 ETAS Contact Addresses

ETAS HQ

ETAS GmbH		
Borsigstraße 14	Phone:	+49 711 89661-0
70469 Stuttgart	Fax:	+49 711 89661-106
Germany	WWW:	www.etas.com

ETAS Subsidiaries and Technical Support

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

ETAS subsidiaries	WWW:	www.etas.com/en/contact.php
ETAS technical support	WWW:	www.etas.com/en/hotlines.php