OsciVerifier

- A prototype to automatically verify the content -

of MDA oscilloscopes

Tutors:

Thomas Gemmi (ETAS)

Prof. Dr. Johannes Maucher (HdM)

Thomas Suchy (HdM)

Document Meta Data:

Document Name:	OsciVerifier
Document Version:	see modification protocol
Owner:	Martin Kemkemer & Alexander Stindl
Project number:	-
Classification	Public
Status:	Released

Filing:

Path and Filename:	https://version.mi.hdm-stuttgart.de/svn/OsciVerifier/trunk/Documentation/OsciVerifier_Documentation.doc
Save Number	1575
Creation Date	13.02.2006
Date of Last Save	2/17/2006 9:29 AM
Last Version Saved By:	Alexander Stindl

Modification protocol:

Document Version	Author	Date	Content (Detailed Modification Protocol)
0.0	Martin Kemkemer & Alexander Stindl	13.02.2006	First version.
1.0	Alexander Stindl	17.02.2006	Review

Open Issues and Pending Decisions (closed issues may be deleted):

OI	What	responsible	

Table of Contents

List of Figures

1 Introduction

OsciVerifier is a cooperation with the Hoschschule der Medien (HdM) and ETAS (Engineering Tools and Application Systems) in Stuttgart. The project is basis of the lecture Software Practical at the HdM. In this lecture students makeup their own projects, test new technologies or as in our case companies like ETAS step up and deliver ideas for interesting projects. In the following chapters we would like to give you a brief overview over our project, the *OsciVerifier*.

1.1 ETAS

The ETAS Group, formed in 2003 through the merger of ETAS, Vetronix, and LiveDevices, today supplies a comprehensive portfolio of standardized development and diagnostic tools that cover the complete development and service life cycles of electronic control units in today's automobiles.

Prior to coming together as the ETAS Group, the three companies had already achieved considerable success in their own rights and their own markets. With their products and services catering to various segments of the automotive embedded systems market, they shared an exclusive focus on the automobile industry and its suppliers. ETAS GmbH is the industry leader in providing a variety of software and hardware development tools for electronic control units in passenger cars and trucks. Closely related and synonymous with the name LiveDevices are the fast and powerful operating systems for automotive microprocessors. Vetronix, by contrast, is positioned differently: Instead of concentrating on the development environment, the company supplies cutting-edge diagnostic tools for vehicle service.

ETAS provides measurement and calibration tools to automakers (OEMs) and suppliers worldwide. ETAS hardware and software products form an integral part of the development process, assisting test engineers in their investigations ranging in scope from vehicle components to functional subsystems or assemblies to entire automobiles.

Engineers use ETAS measurement modules and ECU interfaces to acquire reliable data and calibrate the assemblies. INCA software products are employed in data acquisition, online and offline calibration, and measurement data analysis.

1.2 System Test

Wikipedia: "*System testing is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of Black box testing, and as such, should require no knowledge of the inner design of the code or logic*."

At ETAS the System Test is the last instance of a model called V-Cycle.

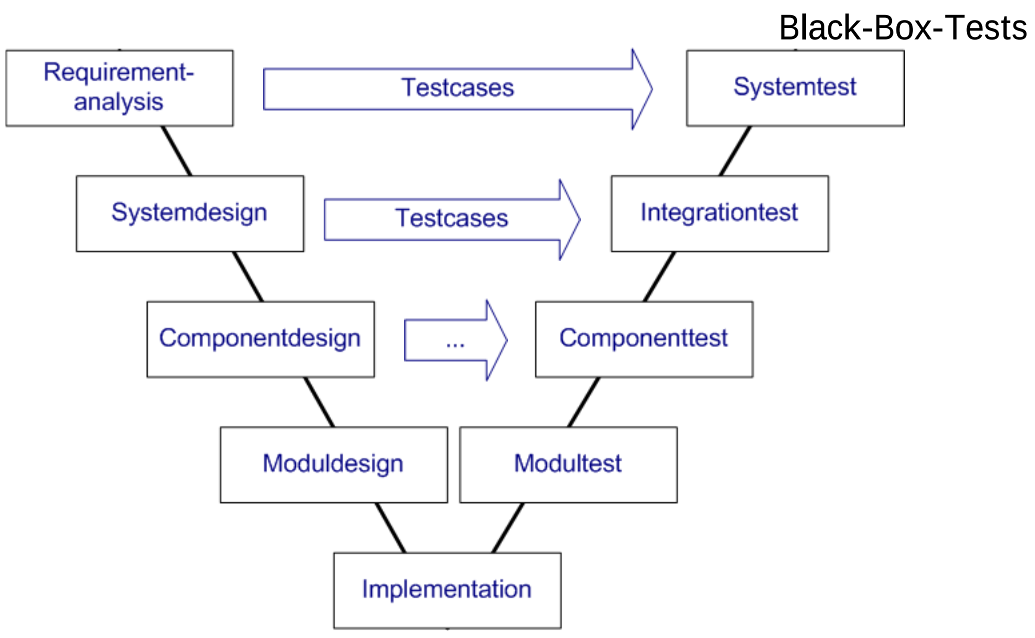


Figure 1: ETAS V-Cycle

As shown in *Figure 1: ETAS V-Cycle* every phase of development (except implementation) has its opponent in testing. This guarantees a comprehensive assurance of quality. For the System Test this means that engineers do not necessarily know about the inner design, the code or logic of the system being tested. Main input information are the requirements that were specified in the analysis. These requirements have to be verified. All steps in between may be disregarded. This proceeding allows a good understanding of the customer's point of view, since the product is tested that way.

Engineers in System Test are supported by tools like *Seque Silktest*. These tools grant automatic execution of test cases and scenarios or automated control of the software being tested. External tools like *OsciVerifier* may be automated as well. Also, these tools have the possibility of computing data such as screenshots or xml files for instance.

1.3 Measure Data Analyzer (MDA)

The Measured Data Analyzer (MDA) program is an offline tool for displaying and analyzing saved measured data. The data is stored in standardized measure files e.g. binary format or timestamp-value relation (ASCII file).

The MDA has two different evaluation windows: the Oscilloscope display window to be used as an oscilloscope and XY-plotter, and the table display that is especially useful for quickly viewing precise values. You can combine measured signals from different measurement files, configure their display on the screen, and save these settings in an evaluation configuration. For the actual analysis, various zoom functions for navigating in the measurement file, a measure cursor for measuring selected values, and automatic difference calculation are available.

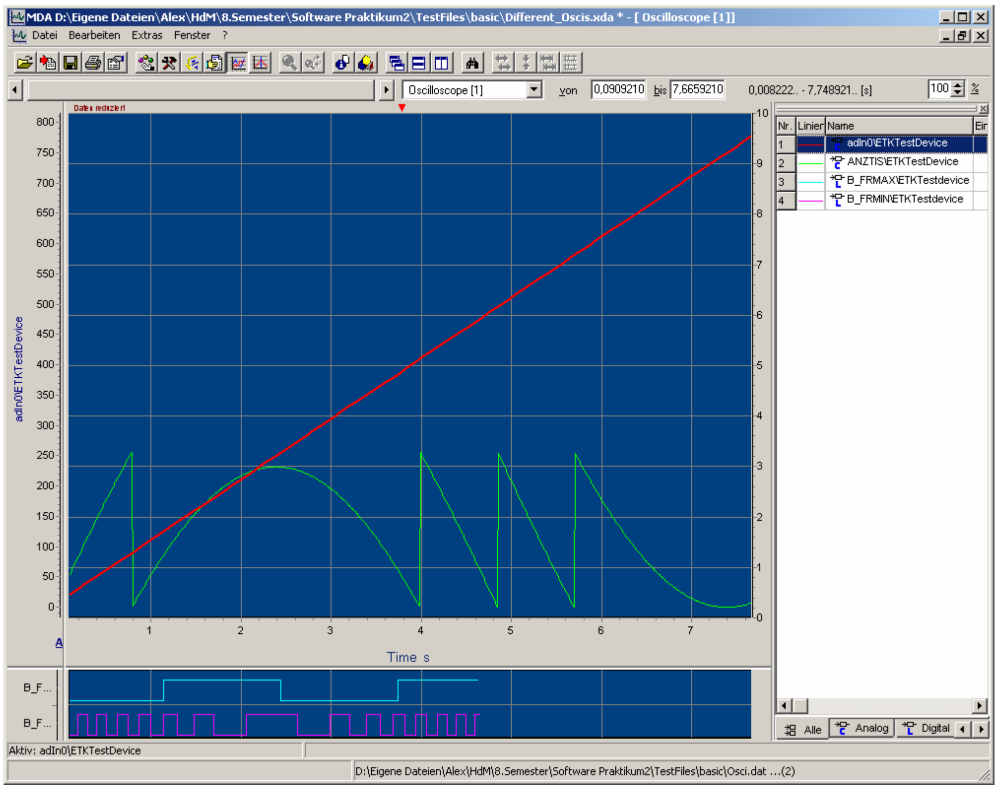

MDA D:\Eigene Dateien\Alex\HdM\8.Semester\Software Praktikum2\TestFiles\basic\Different_Oscis.xda * - [Oscilloscope [1]]
Datei Bearbeiten Extras Fenster ?
Oscilloscope [1]
von 0,0909210 bis 7,6659210
0,008222.. - 7,748921.. [s]
100
adIn0\ETKTestDevice
Time s
Nr. Linien Name
1 adIn0\ETKTestDevice
2 ANZTIS\ETKTestDevice
3 B_FRMAX\ETKTestdevice
4 B_FRMIN\ETKTestdevice
Alle Analog Digital
Aktiv: adIn0\ETKTestDevice
D:\Eigene Dateien\Alex\HdM\8.Semester\Software Praktikum2\TestFiles\basic\Osci.dat ...(2)

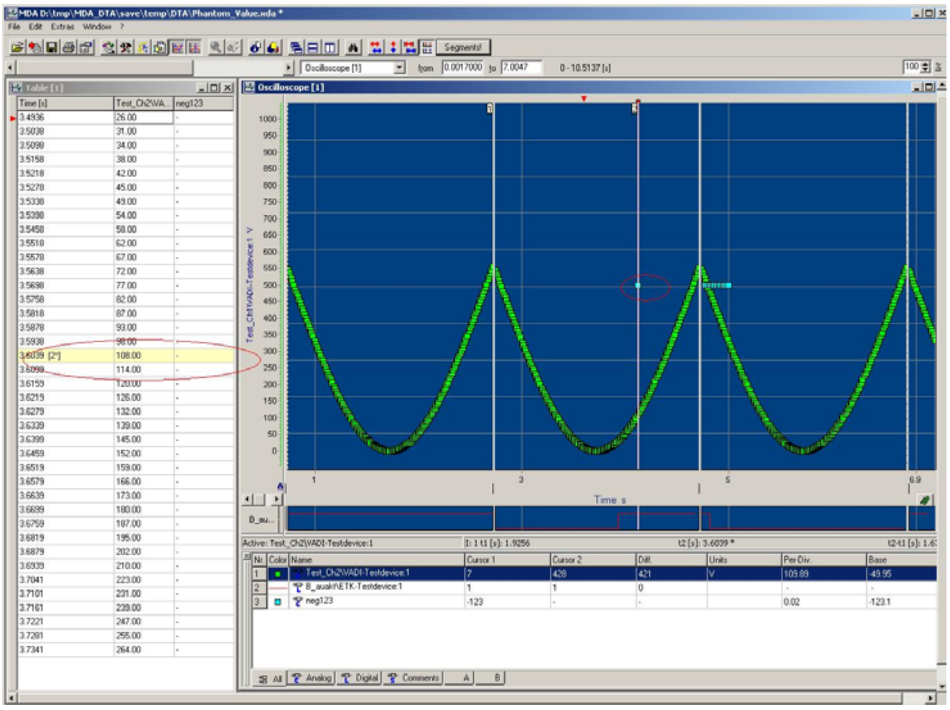
Figure 2: Measure Data Analyzer (MDA)

In the MDA several oscilloscopes can be displayed. In *Figure 2: Measure Data Analyzer (MDA)* there are two oscilloscopes. The first oscilloscope displays analog signals and the one below digital ones. The different types of oscilloscopes will become part of the requirements later on.

Another aspect on the MDA is that signals, especially the analog signals, may be displayed in different modes such as step, line and timestamp mode. These modes will also be described later.

1.4 The Problem

As a matter of fact in the MDA rendering errors may occur in an oscilloscope. So, how can you verify the content of the oscilloscopes of the MDA, before a possible error may occur on customer side? Manual testing is not an appropriate solution, since it is too time consuming. Present tools like “BeyondCompare” are insufficient, because they depend on the display resolution of the test system and have no back-tracking of failures. In addition there is no appropriate test of identifying these errors. Therefore an additional tool will be required: *OsciVerifier*.


MDA D:\tmp\MDA_DTA\save\temp\DTA\Phantom_Value.xda *
File Edit Extras Window ?
Segments!
Oscilloscope [1]
from 0.0017000 to 7.0047
0 - 10.5137 [s]
Table [1]
Oscilloscope [1]
Test_Ch1\VADI-Testdevice:1 V
Time s
D_au...

Time [s]	Test_Ch2\VA...	neg123
3.4936	26.00	-
3.5038	31.00	-
3.5098	34.00	-
3.5158	38.00	-
3.5218	42.00	-
3.5278	45.00	-
3.5338	49.00	-
3.5398	54.00	-
3.5458	58.00	-
3.5518	62.00	-
3.5578	67.00	-
3.5638	72.00	-
3.5698	77.00	-
3.5758	82.00	-
3.5818	87.00	-
3.5878	93.00	-
3.5938	98.00	-
3.6039 [2*]	108.00	-
3.6099	114.00	-
3.6159	120.00	-
3.6219	126.00	-
3.6279	132.00	-
3.6339	139.00	-
3.6399	145.00	-
3.6459	152.00	-
3.6519	159.00	-
3.6579	166.00	-
3.6639	173.00	-
3.6699	180.00	-
3.6759	187.00	-
3.6819	195.00	-
3.6879	202.00	-
3.6939	210.00	-
3.7041	223.00	-
3.7101	231.00	-
3.7161	239.00	-
3.7221	247.00	-
3.7281	255.00	-
3.7341	264.00	-

Active: Test_Ch2\VADI-Testdevice:1 I: 1 t1 [s]: 1.9256 t2 [s]: 3.6039 * t2-t1 [s]: 1.6

Nr.	Color	Name	Cursor 1	Cursor 2	Diff.	Units	Per Div.	Base
1	🟩	Test_Ch2\VADI-Testdevice:1	7	428	421	V	109.89	-49.95
2	—	B_auak\ETK-Testdevice:1	1	1	0		-	-
3	🟦	neg123	-123	-	-		0.02	-123.1

Figure 3: Rendering error in the MDA

In *Figure 3: Rendering error in the MDA*, a value is displayed in the analog oscilloscope, which does not exist in the measurement file. This should not happen!

1.5 Requirements

ETAS gave us a list of user problems, which had to be solved by OsciVerifier. Therefore we created our own requirements out of this list of user problems (see *2 Project Management* for more details). The main requirements are:

- Automated search for failures in oscilloscope of MDA for the different kind of signals (analog and digital)
- Define communication between *OsciVerifier* and Silktest. It should be possible to exchange information like input parameters, test results etc.
- Operate *OsciVerifier* via command line
- A back tracking of failures must be possible (e.g. timestamp or time area of failure)
- *OsciVerifier* must support different visualization modes for waveforms. These modes are: line, timestamp and step mode

All functionality of *OsciVerifier* is based on these requirements.

2 Project Management

Besides designing and implementing, there was also a focus on managing the project. Our first task was to create a rough project plan, which did not have a completed and fully detailed overview at first but already had the time span of *OsciVerifier* and certain fixed dates like the project start, several status meetings (no milestones yet) and the MediaNight. Also, the plan indicated the certain project phases (*see 2.1 Project Plan for more details*).

In weekly status meetings at ETAS the progress of the project was discussed and decisions were made for the further development of the project. Since ETAS was our "customer", these status meetings were necessary not to lose focus and stick to the customer's user problems. Decisions and assigned tasks were documented in a status report or sometimes in a meeting minutes excel sheet

In order to keep everybody, who was involved in the project, informed, a status report was sent every week. The report had three parts. In the first part, everything that was done or problems that occurred in the week before were described. The second part contained decisions or appointments that were made in status or team meetings. The last part was an overview on what was going to happen in the next couple of weeks.

Another task of project management was to create our own requirements of the prototype out of a list of user problems from ETAS. In this list ETAS pointed out several problems, which *OsciVerifier* had to solve (*Please refer to the document "List_of_UPs_requirements.doc" for more details*). These requirements were marked with an ID similar to the number of the user problem and ranked with a priority, whereas "1" means the highest priority.

An example:

***ID:** UP0001*

***Description of the problem**: Automated search for failures in visualization component (oscilloscope) of MDA for continuous signals.*

***List of requirements**:*

- *Read in data files and generate legal Pixels with appropriate math algorithms*
- *Read in MDA oscilloscope from screenshot and compare received signal pixels with the legal pixels.*
- *Search for errors*
- *back track failures*
- *generate report file (.xml)*

***Priority**: 1*

Last but not least all project documentation (project plan, UML, JavaDoc, milestone presentations …) had to be done in English, which was required by ETAS.

2.1 Project Plan

In a second step, a more detailed plan was constructed, based on the previous one. The new plan, which was going to be our guideline till the end of the project, then contained all information, which was necessary to complete the project successfully. And, despite of a few appointments of status meetings, which did not take place or had to be shifted to a later date, like the milestone presentation at ETAS in front System Test experts, all phases and fixed dates were planed and accomplished nearly in time.

How did we plan?

First we inserted the fixed dates like the project start on October 18th, the MediaNight on January 26th as possible project end and the status meetings with ETAS, which took place on nearly every Tuesday. Then we calculated backwards from the project end. We estimated an implementation time of 70 hours per student, which could be done in 7 weeks. Important to us was the design phase, for what we calculated 3 weeks. The remaining 4 weeks were separated in 3 weeks of creating a Feasibility study and one week of approving the concept, chosen in the feasibility phase.

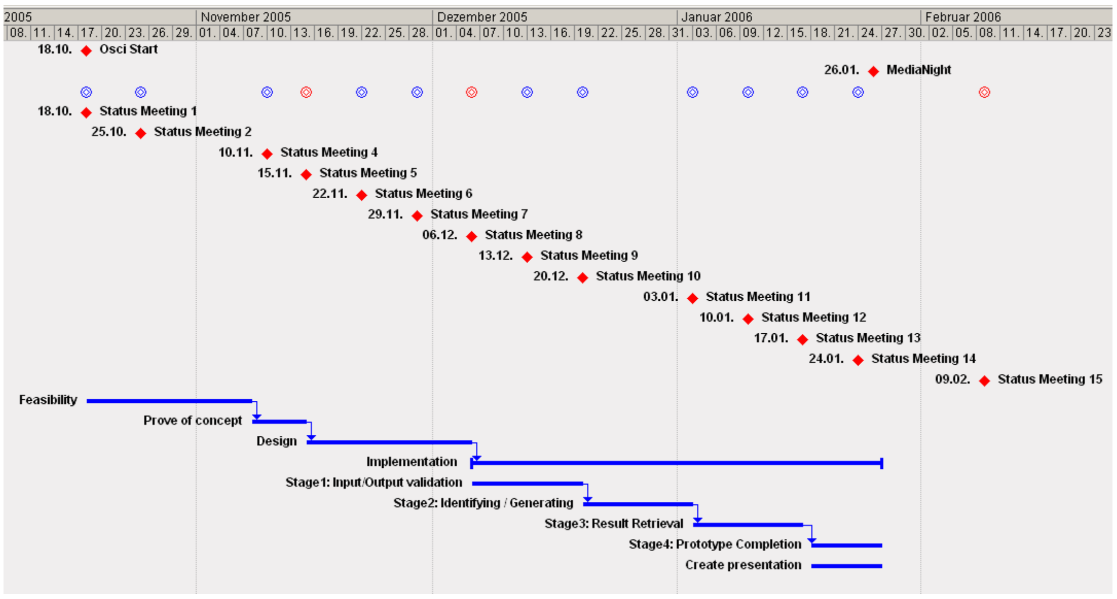


Figure 4: Project Plan

Note that important milestones are marked with two red circles (◎). On these dates the individual presentations were held. The marker for the presentation at the HdM ("MI-Präsentationstag") misses in this example, but may be supplemented by the MediaNight marker, since the presentation day at the HdM is always the day before the MediaNight.

The individual project phases:

- Feasibility:

  Several strategies of solving the problem (*to automatically verify the content of MDA oscilloscopes*) had to be found and evaluated. In total we received three methods, which were capable. The first method ("Graphical") would generate a mask out of the measurement file and identify an error in the overlay of the two pictures. The second method ("Measured Data") basically did the same, but instead of overlaying the masks, two arrays of pixels, which were read from the two pictures, were compared. An error would be found, when there is a pixel value in one array, but not in the other.

  The third method ("Mathematical"), our chosen one, also compared two arrays of pixels, but did not create a mask to read pixels from. Instead these pixels are generated directly from the measurement file using appropriate math algorithms. This method will be described in detail later on. Please see the document "01_Feasibility_Study.ppt" for more information on the other methods.

- Prove of concept:

  This phase was necessary to check for possible risks that may occur during the project and if the chosen method from the feasibility phase is even practicable. As we chose Java as programming language, we had to check for the legal rights of certain libraries (e.g. JDOM), since ETAS is going to use *OsciVerifier* in the System Test. Possible risks for example were that the needed math algorithms may not apply or are too hard to implement and if there is a way to calculate between measurement values and pixel values, which would guarantee a back tracking of failures.

  We decided that the risks are minimal and the solution should work. Also, there are no legal problems with the used Java libraries.

- Design:

  This phase was very important to us, since we wanted to avoid as much error potential as possible. Also, it was necessary to divide the tasks of implementing in an appropriate and well known way. Without designing we would have had a hard time of knowing who is implementing exactly what.

  We did 2 weeks of designing and accomplished following diagrams:

  - Use Case Diagram
  - State Diagram
  - Activity Diagram
  - Class Diagram

  These diagrams helped us not to lose focus. They are described in an extra chapter (*4 Design*).

- Implementation:

  The implementation phase was subdivided into 4 stages. These stages were part of the requirement of stage delivery, which means that we had to deliver our sources to ETAS after every stage for testing purposes:

- Stage1: Input/Output validation

  This stage implemented the read in of all input files (input.xml, measurement file, signal screenshot and reference screenshot).

- Stage2: Identifying / Generating

  In this stage, we had to generate legal pixels out of the measurement file and to identify the different signals and oscilloscopes (analog & digital) from the given screenshots. The identified signals had to be saved in signal pixels.

- Stage3: Result Retrieval

  After Identifying / Generating we had to compare the two arrays of pixels (legal & signal pixels) in order to find a possible error. Also, the time of the error had to be qualified as well as a result file needed to be written.

- Stage4: Prototype Completion

  Prototype Completion meant that we had to test prototype functionality, rework possible varieties in the diagrams from design phase and complete JavaDoc and source code documentation.

- Create presentation:

  This phase reminded us to keep a time buffer to create a presentation for the MediaNight and ETAS.

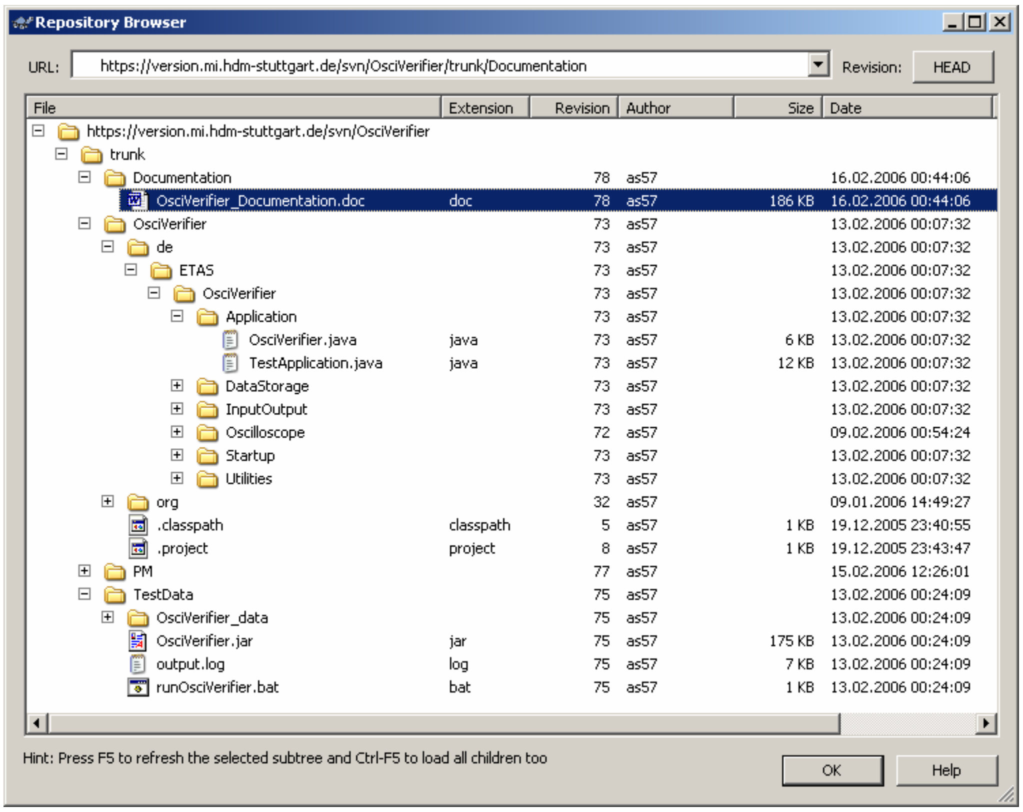
2.2 Milestone presentations

At the end of every important phase a milestone presentation was held. All together there were four presentations:

- Feasibility Study: All methods of solving the problem were described and rated (Pros & Cons) in this presentation.
- Design: All diagrams, created in the design phase were presented. Also, first implementations from *Stage1: Input/Output validation* were demonstrated.
- MediaNight: A brief overview over the whole project was presented at the “MI-Präsentationstag” at the HdM.
- At ETAS: A more detailed and more technical presentation than the one for the MediaNight with a good insight in the architecture of the project. It lasted about 45 minutes and was held in front of ETAS System Test experts.

2.3 Repository Structure of OsciVerifier

In order to guarantee a smooth proceeding in the different phases of the project, especially in the implementation phase, we decided to use the HdM's subversion server to keep a clear repository structure of *OsciVerifier*. So, everybody of the team always had the actual project data, and no one had to merge source code files or build new versions of the tool.



URL: https://version.mi.hdm-stuttgart.de/svn/OsciVerifier/trunk/Documentation Revision: HEAD

File	Extension	Revision	Author	Size	Date
https://version.mi.hdm-stuttgart.de/svn/OsciVerifier					
trunk					
Documentation		78	as57		16.02.2006 00:44:06
OsciVerifier_Documentation.doc	doc	78	as57	186 KB	16.02.2006 00:44:06
OsciVerifier		73	as57		13.02.2006 00:07:32
de		73	as57		13.02.2006 00:07:32
ETAS		73	as57		13.02.2006 00:07:32
OsciVerifier		73	as57		13.02.2006 00:07:32
Application		73	as57		13.02.2006 00:07:32
OsciVerifier.java	java	73	as57	6 KB	13.02.2006 00:07:32
TestApplication.java	java	73	as57	12 KB	13.02.2006 00:07:32
DataStorage		73	as57		13.02.2006 00:07:32
InputOutput		73	as57		13.02.2006 00:07:32
Oscilloscope		72	as57		09.02.2006 00:54:24
Startup		73	as57		13.02.2006 00:07:32
Utilities		73	as57		13.02.2006 00:07:32
org		32	as57		09.01.2006 14:49:27
.classpath	classpath	5	as57	1 KB	19.12.2005 23:40:55
.project	project	8	as57	1 KB	19.12.2005 23:43:47
PM		77	as57		15.02.2006 12:26:01
TestData		75	as57		13.02.2006 00:24:09
OsciVerifier_data		75	as57		13.02.2006 00:24:09
OsciVerifier.jar	jar	75	as57	175 KB	13.02.2006 00:24:09
output.log	log	75	as57	7 KB	13.02.2006 00:24:09
runOsciVerifier.bat	bat	75	as57	1 KB	13.02.2006 00:24:09

Hint: Press F5 to refresh the selected subtree and Ctrl-F5 to load all children too

Figure 5: Subversion repository

As shown in *Figure 5: Subversion repository* there are four folders below the trunk of the repository. The documentation of *OsciVerifier* lies in the "Documentation" folder. All sources of *OsciVerifier* are in the "OsciVerifier" folder. The "PM" folder contains the project plan and the milestone presentations. The last folder "TestData" includes real test data, an executable JAR file of *OsciVerifier* and a batch job to start it. This example was demonstrated at the MediaNight and has a built in error in one of the given screenshots that is supposed to be found with the tool.

3 Analysis & Specification

Before we could start with the programming of *OsciVerifier* we had to analyze and specify certain procedures. In short, we had to find answers to following questions:

- What kind of input information is necessary?
- How is this information provided?
- How is *OsciVerifier* started or how does it communicate with other applications?
- What about Logging and Reporting?

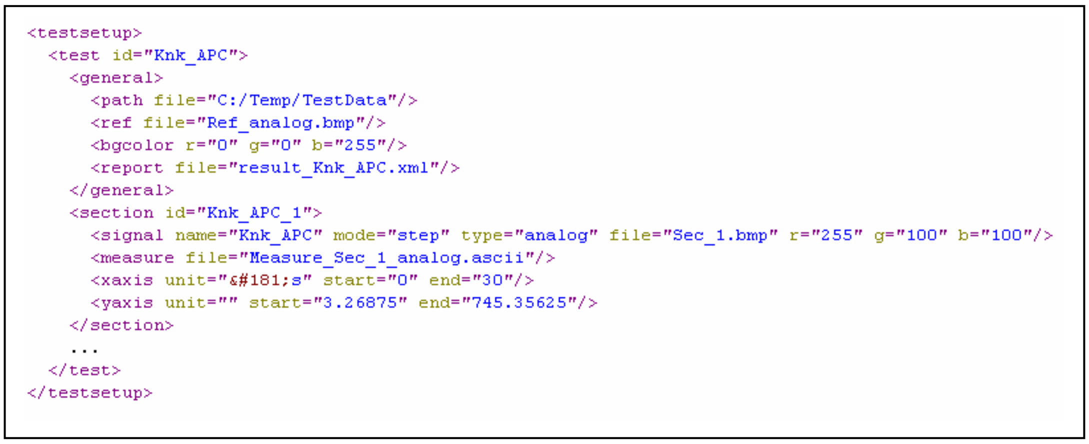
These questions will be answered in the following chapters.

3.1 Input information

There is quite a lot of information required for the automated verification of an oscilloscope. *OsciVerifier* needs two screenshots, one to identify the oscilloscope and one to identify the signal inside of the oscilloscope. Both, the oscilloscope and the signals, are identified by their color, therefore this information is required as well.

In order to generate legal pixels, the measurement file is necessary as well. Also, *OsciVerifier* needs the location of these test files in the file system to able to find them. For the result file information like the unit of the axis, the report location and an explicit test ID are interesting. To achieve a back tracking of failures, the exact start and end values of the axis are valuable.

All this information is provided and read by *Silktest*, which generates a xml file:



```xml
<testsetup>
  <test id="Knk_APC">
    <general>
      <path file="C:/Temp/TestData"/>
      <ref file="Ref_analog.bmp"/>
      <bgcolor r="0" g="0" b="255"/>
      <report file="result_Knk_APC.xml"/>
    </general>
    <section id="Knk_APC_1">
      <signal name="Knk_APC" mode="step" type="analog" file="Sec_1.bmp" r="255" g="100" b="100"/>
      <measure file="Measure_Sec_1_analog.ascii"/>
      <xaxis unit="&#181;s" start="0" end="30"/>
      <yaxis unit="" start="3.26875" end="745.35625"/>
    </section>
    ...
  </test>
</testsetup>
```

Figure 6: input.xml

This xml file is a required startup parameter. *OsciVerifier* parses the file and stores all the information in corresponding Java classes. Note, that there is a TestSetup, Test and Section class in *OsciVerifier* just like in the xml file.

3.2 OsciVerifer startup

OsciVerifier is started over the command line. Therefore all sources were packed in an executable JAR file, which may be started with certain parameters in a batch job:

```
java -Xms200M -Xmx400M -jar OsciVerifier.jar
"C:/Temp/TestData/input.xml" -tolerance:2 -debugMode:ON > output.log
```

Since we had problems with the large amount of data (one screenshot with a size of 2MB) being inspected, we had to raise the minimum and maximum heap size of the Java Virtual Machine, which is done by using the parameters `-Xms` and `–Xmx`.

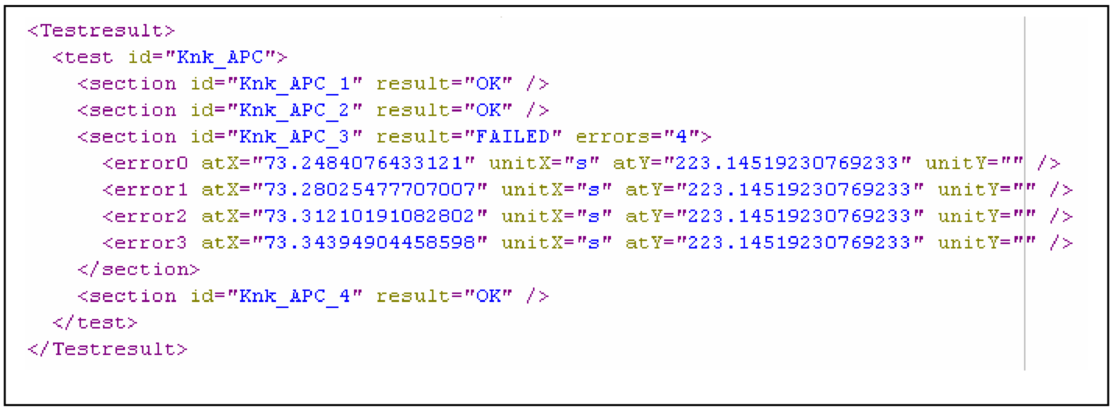
The first parameter after the JAR file, which is initialized with `–jar`, delivers the location and name of the input.xml file. To define a tolerance, in which the specific signal pixels have to lie, the parameter `-tolerance:x` is used. Also, a debugging mode is implemented, which will display possible errors graphically. This allows testers to reproduce certain test cases by hand and verify the corresponding result. To turn on debug mode use `-debugMode:ON`. The parameters of tolerance and debugMode may be omitted, since they both have default values (“4” for the tolerance and “OFF” for debugMode).

The Pipe “`> output.log`“ captures the logging information printed out by *OsciVerifier* in a LOG file.

3.3 Logging & Reporting

Logging is done by capturing the output of *OsciVerifier* in a LOG file as mentioned above.

A more important feature is the reporting. As a report, *OsciVerifier* generates a xml file, listing result information for every test run. There is always one report file for every test in a test setup.



```
<Testresult>
  <test id="Knk_APC">
    <section id="Knk_APC_1" result="OK" />
    <section id="Knk_APC_2" result="OK" />
    <section id="Knk_APC_3" result="FAILED" errors="4">
      <error0 atX="73.2484076433121" unitX="s" atY="223.14519230769233" unitY="" />
      <error1 atX="73.28025477707007" unitX="s" atY="223.14519230769233" unitY="" />
      <error2 atX="73.31210191082802" unitX="s" atY="223.14519230769233" unitY="" />
      <error3 atX="73.34394904458598" unitX="s" atY="223.14519230769233" unitY="" />
    </section>
    <section id="Knk_APC_4" result="OK" />
  </test>
</Testresult>
```

Figure 7: report.xml

As shown in *Figure 7:report.xml* for every section in a test there is a result. If any errors occur, the total amount of errors and detailed information of the first 100 errors are listed for the sorresponding section. In order to still receive some report information in case of a system crash, *OsciVerifier* writes and overwrites the report file every time a section has finished.

4 Design

The feasibility phase was used to prove different methods to verify the content of an oscilloscope.

In the design phase the system components and their interaction were specified. The chosen method needed to be transformed in a system design. We used the Borland Together Architect 2006 to create different kinds of UML 2.0 diagrams.

At this point the method gets explained so that the context between method and diagrams can be understood.

4.1 The method

In the feasibility phase we thought about different methods to verify the content of an oscilloscope. We came up with three versions to solve the main challenge.

- Method 1 (graphical): Overlay of two oscilloscope pictures
- Method 2 (measure data): Compare measure data
- Method 3 (mathematical) Generate table of legal pixels

All of the three methods of course had different advantages and disadvantages. At this point I will only talk about the proceeding of method 3, which we decided to implement.

The most important advantage of this version was that there was no necessity to build an additional mask. Unlike method 1 and method 2, the mathematical version was not using any kind of picture comparison. By not creating a reference oscilloscope on the host machine to be compared with some screenshot taken by another machine, we avoided dependencies on the graphical environments e.g. display resolution.

Generate table of legal pixels

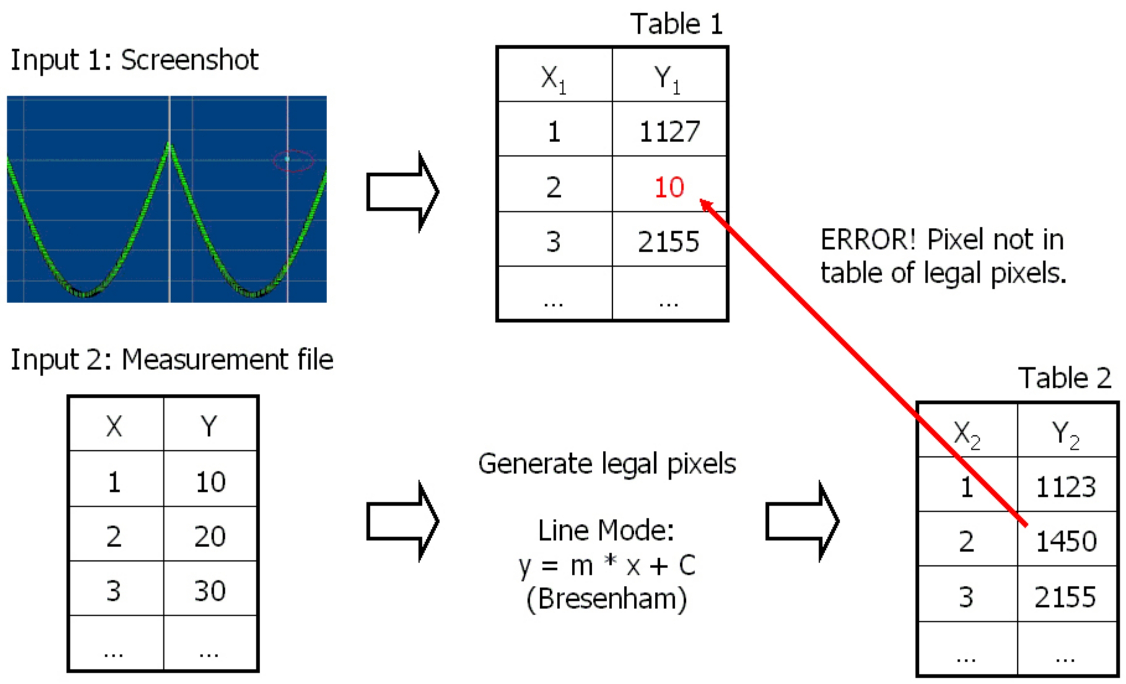


Table 1

X_1	Y_1
1	1127
2	10
3	2155
…	…

Input 2: Measurement file

X	Y
1	10
2	20
3	30
…	…

Table 2

X_2	Y_2
1	1123
2	1450
3	2155
…	…

Figure 8: mathematical method

Proceeding:

Like in all of the methods there are two input documents to be used.

The first input is the screenshot taken from the MDA. This screenshot holds the oscilloscope with the specified signal information generated by the MDA graphic library. It therefore shows the possible errors to be found.

Second input document is an ASCII measurement file. There is a single file for every signal in any oscilloscope. This file holds the real measurement data. The data is assumed to be the correct data to be drawn.

The first step is to identify the relevant oscilloscopes. The information to be saved about every oscilloscope is the starting point (upper left corner) as well as the x- and y-size. With this data it is now possible to identify and read the actual signal data values. The pixel values of the signal file are stored in an ArrayList. The ArrayList corresponds to “Table 1” in the above diagram.

Second step is the generation of the legal pixels. We use the measurement file data to generate the corresponding pixel data. This is achieved by using different math algorithms. The *OsciVerifier* implements three different modes. They are going to be described in detail later. The generated pixel data is stored in an ArrayList which is displayed as “Table 2” in the above diagram.

To get the actual error pixels, the two ArrayList are now compared. Every signal pixel which has no corresponding legal pixel is an error pixel. There is a tolerance used to compensate possible rounding errors and a signal file which is usually two pixels thick. This tolerance expands the legal pixels e.g. with a tolerance of 4 not only the legal pixel but every pixel in a range of 4 pixels is recognized to be ok.

After identifying the error pixels, the data gets scaled back to the measurement scale. Like this you can later easily identify the error in the corresponding oscilloscope. When taking a short look at the relevant oscilloscope, no one is willing to work with the pixel coordinates the error was found at.

Advantages:

- Very accurate
- Flexible to any kind of input (e.g. resolution, peaks, pauses, etc.)
- No need to create an additional mask to compare two screenshots
- Approved mathematic algorithms

Disadvantages:

- Large amount of data has to be processed and stored

4.2 A way towards implementation

To gain a structural approach towards implementation and get a basis for discussion it was necessary to work out some UML conform diagrams. The diagrams turned out to be very useful with regard to any discussion about the tasks of the different classes or components, potential changes in architecture or just the partitioning of the code to be implemented.

Also our first design did not match the later implementation in detail, the main structure was maintained. For documentation purposes there are two class diagrams shown. The first one is the one worked out in the actual design phase, the other one is gained by reengineering the code at the end of implementation.

4.2.1 Use Case Diagram

The Use Case Diagram is a technique used to record the functional system requirements. They describe the typical interactions between the system user and the system itself. They explain how a system is to be used.

The Application SilkTester is modified as the only SystemUser. All it can do is just “run the *OsciVerifier*”. We see that this Use Case includes some other Use Cases like “process Test” or “read input XML”. Actually this Use Case Diagram was the first diagram to be drawn, it includes too much information about the system itself. This system information is better kept in the Activity Diagram as we see later on.

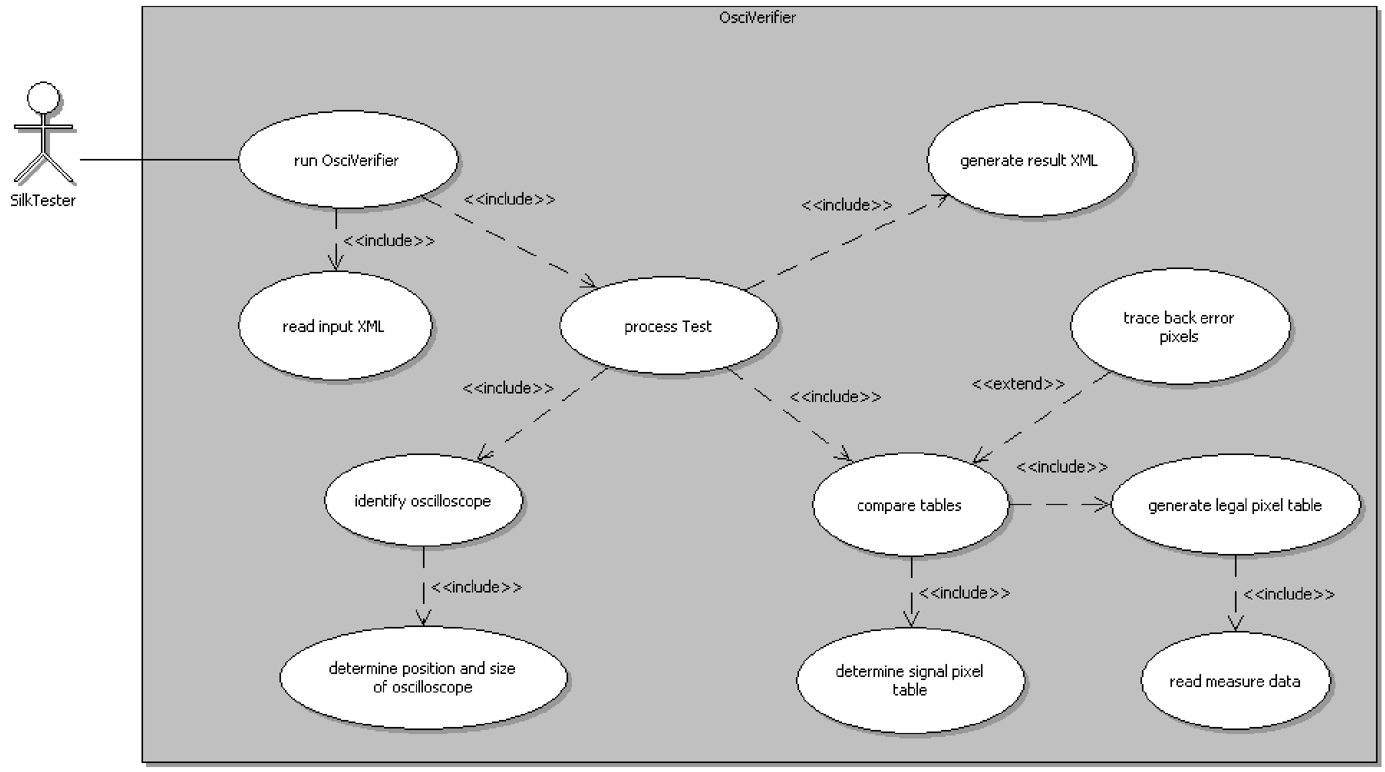


Figure 9: Use Case Diagram

4.2.2 State Machine Diagram

State Machine Diagrams are a usual technique to describe the behavior of a system.

The *OsciVerifier* knows two main states, which are idle and active. Default mode is idle. After starting the *OsciVerifier* its state changes from idle to active. It then can pass through a couple of sub states. In the diagram the *OsciVerifier* executes all of the test cases (tests) and writes a final result ASCII file, before it shuts down.

In the actual implementation of the *OsciVerifier* we decided to write a result xml file after every test. This has the advantage that in the case of an anticipated system breakdown, at least the test results that have been processed until the breakdown are saved.

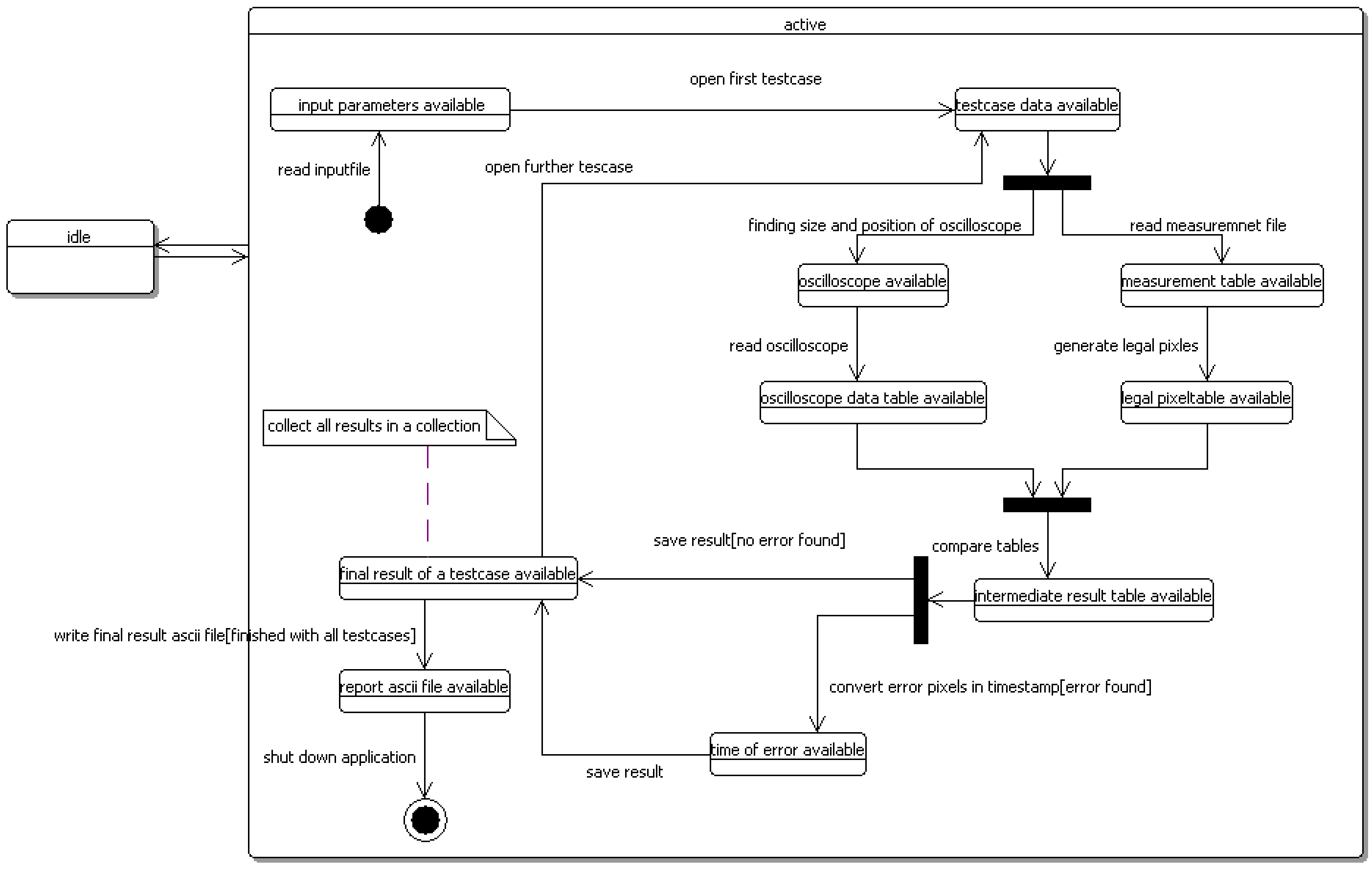

active
idle
input parameters available
open first testcase
testcase data available
read inputfile
open further tescase
finding size and position of oscilloscope
read measuremnet file
oscilloscope available
measurement table available
read oscilloscope
generate legal pixles
oscilloscope data table available
legal pixeltable available
collect all results in a collection
save result[no error found]
compare tables
final result of a testcase available
intermediate result table available
write final result ascii file[finished with all testcases]
report ascii file available
convert error pixels in timestamp[error found]
shut down application
time of error available
save result

Figure 10: State Chart Diagram

4.2.3 Activity Diagram

Activity Diagrams are a technique to describe procedural logic, business processes and workflows. In contrast to flowcharts they are able to display parallel behavior.

This Activity Diagram was created for presentation purposes after finishing the implementation. It describes the procedural logic of the *OsciVerifier*.

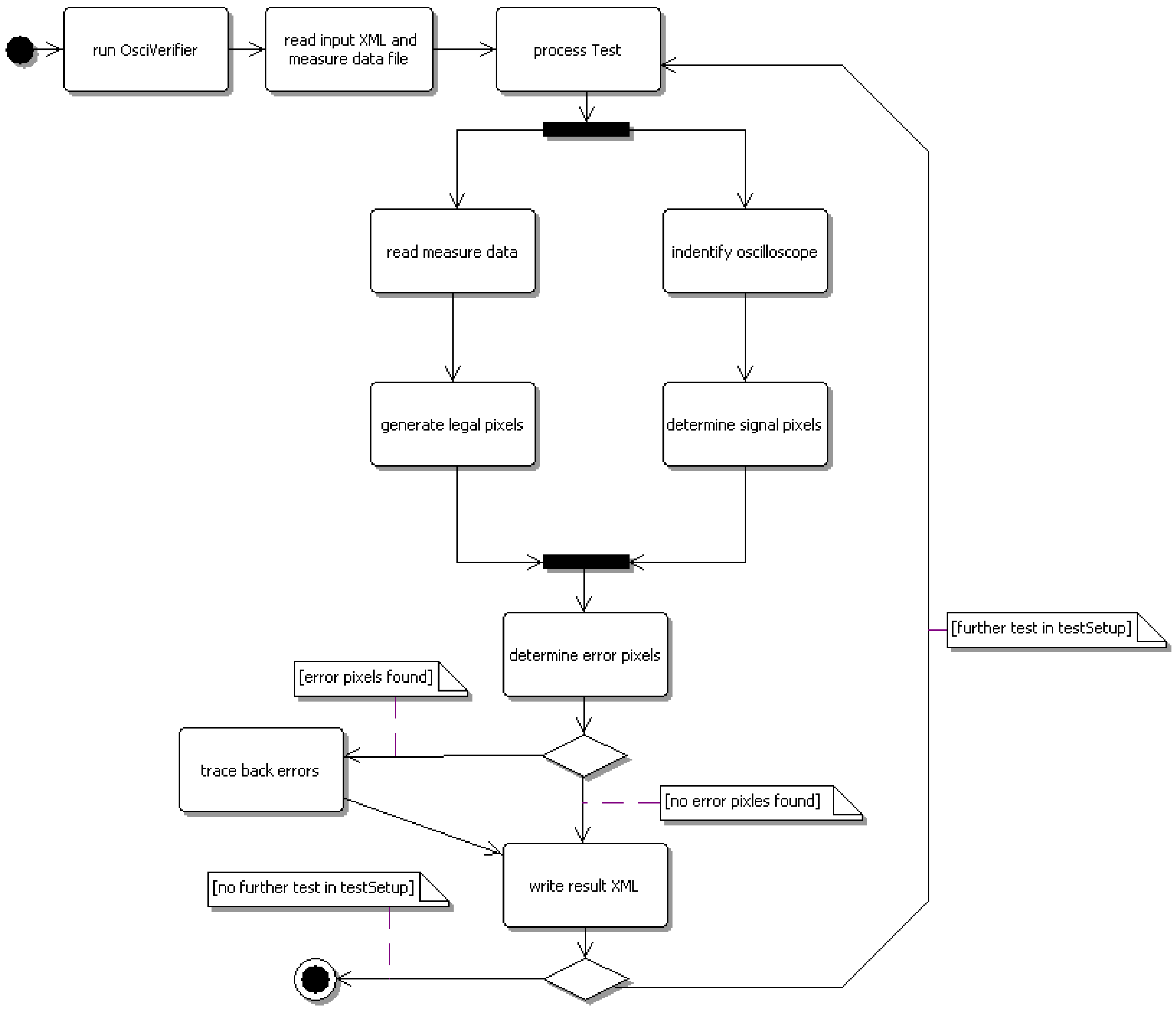


Figure 11: Activity Diagram

4.2.4 Class Diagram

The Class Diagram is the most common UML diagram. Class Diagrams describe the object types in the system and the different static relations between them. In addition they show the features and operations of a class and tell you what kinds of restrictions are existent for object relations

The first Class Diagram shows the system as specified at the beginning of the project. Some of the classes and operations were not used in the later implementation, but overall the design tended to simply the system. The second diagram, which is reengineered is more complex and includes a couple of classes with functionality we were not aware of at the very beginning.

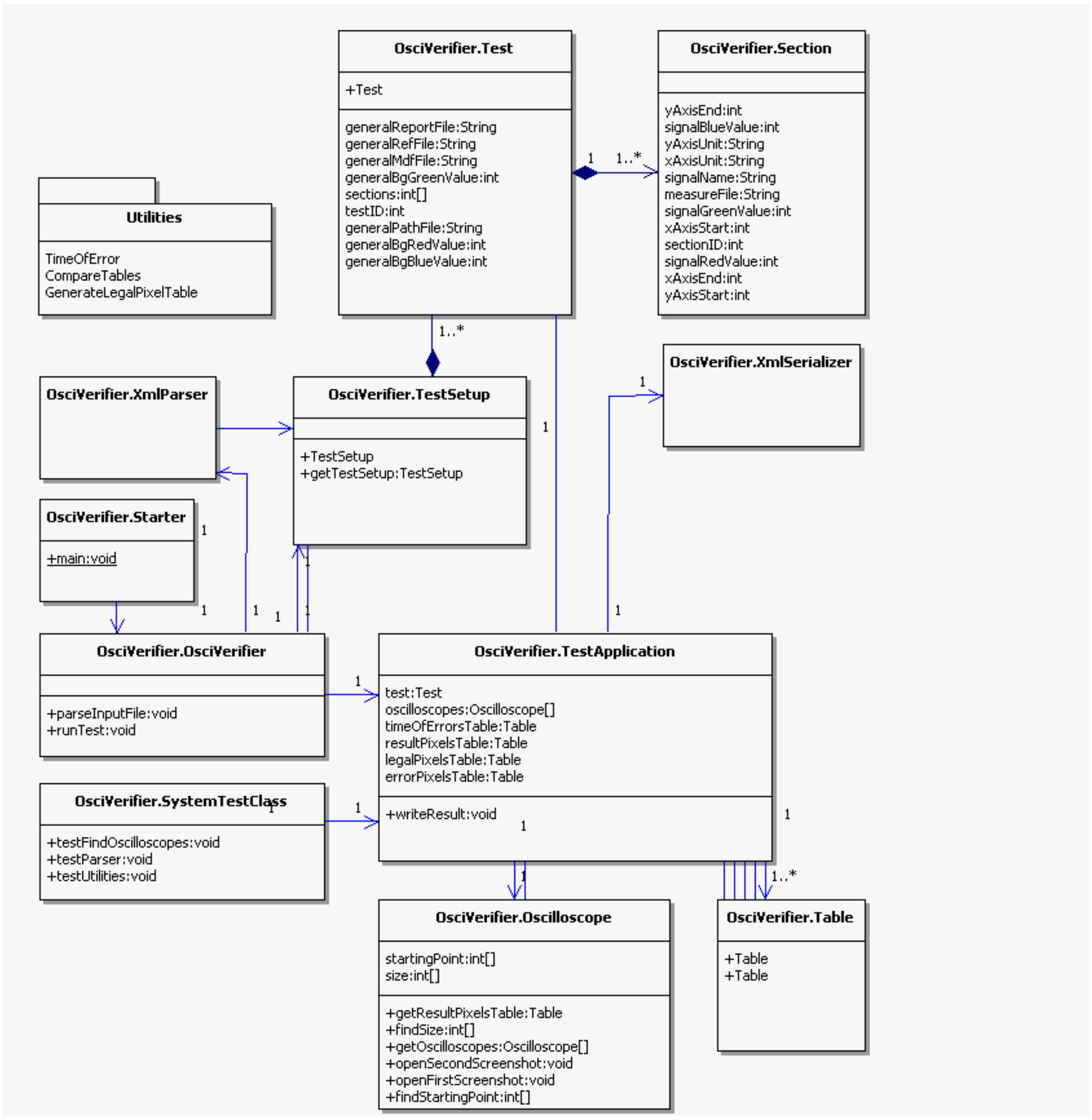


Figure 12: Class Diagram in design phase

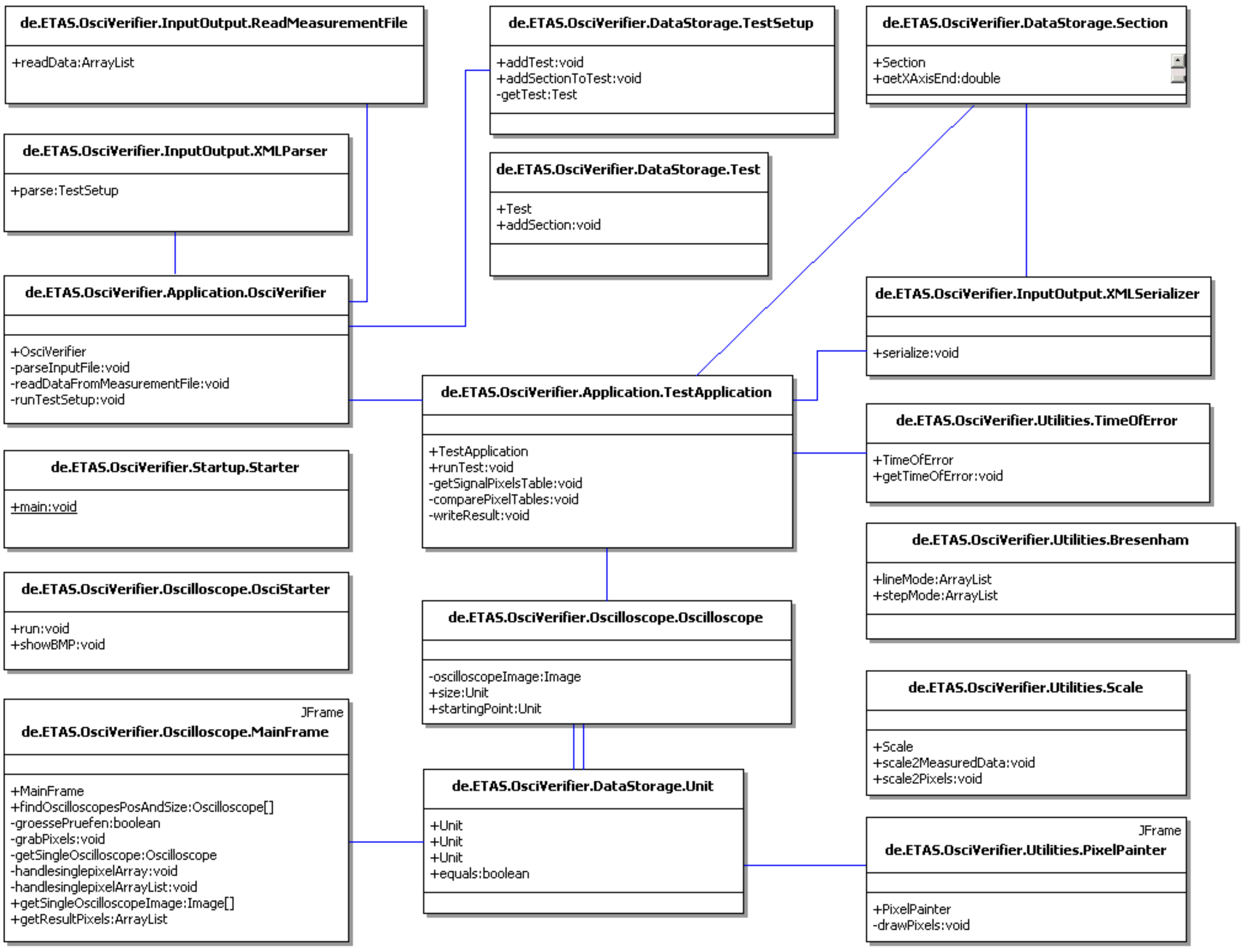

de.ETAS.OsciVerifier.InputOutput.ReadMeasurementFile
+readData:ArrayList
de.ETAS.OsciVerifier.DataStorage.TestSetup
+addTest:void
+addSectionToTest:void
-getTest:Test
de.ETAS.OsciVerifier.DataStorage.Section
+Section
+getXAxisEnd:double
de.ETAS.OsciVerifier.InputOutput.XMLParser
+parse:TestSetup
de.ETAS.OsciVerifier.DataStorage.Test
+Test
+addSection:void
de.ETAS.OsciVerifier.Application.OsciVerifier
+OsciVerifier
-parseInputFile:void
-readDataFromMeasurementFile:void
-runTestSetup:void
de.ETAS.OsciVerifier.InputOutput.XMLSerializer
+serialize:void
de.ETAS.OsciVerifier.Application.TestApplication
+TestApplication
+runTest:void
-getSignalPixelsTable:void
-comparePixelTables:void
-writeResult:void
de.ETAS.OsciVerifier.Utilities.TimeOfError
+TimeOfError
+getTimeOfError:void
de.ETAS.OsciVerifier.Startup.Starter
+main:void
de.ETAS.OsciVerifier.Utilities.Bresenham
+lineMode:ArrayList
+stepMode:ArrayList
de.ETAS.OsciVerifier.Oscilloscope.OsciStarter
+run:void
+showBMP:void
de.ETAS.OsciVerifier.Oscilloscope.Oscilloscope
-oscilloscopeImage:Image
+size:Unit
+startingPoint:Unit
de.ETAS.OsciVerifier.Utilities.Scale
+Scale
+scale2MeasuredData:void
+scale2Pixels:void
JFrame
de.ETAS.OsciVerifier.Oscilloscope.MainFrame
+MainFrame
+findOscilloscopesPosAndSize:Oscilloscope[]
-groessePruefen:boolean
-grabPixels:void
-getSingleOscilloscope:Oscilloscope
-handlesinglepixelArray:void
-handlesinglepixelArrayList:void
+getSingleOscilloscopeImage:Image[]
+getResultPixels:ArrayList
de.ETAS.OsciVerifier.DataStorage.Unit
+Unit
+Unit
+Unit
+equals:boolean
JFrame
de.ETAS.OsciVerifier.Utilities.PixelPainter
+PixelPainter
-drawPixels:void

Figure 13: Class Diagram re-engineered

5 Implementation

OsciVerifier was implemented in Java. Implementation Environment was Eclipse 3.1. The Code was structured into 6 packages, with a total amount of 18 classes and about 2600 lines of code.

5.1 Identifying

Both oscilloscope as well as signal pixels are identified by grabbing all of the pixels in the screenshot. The needed pixels are filtered and stored in either an array or an array list.

<u>Identifying the different kinds of oscilloscopes:</u>

To identify the different kinds of oscilloscope the *OsciVerifier* uses a given reference screenshot of the MDA for every test. This means that every section in a test refers to the same reference screenshot, but also has an own equivalent screenshot of the MDA (due to position and size of the oscilloscopes) with the specified signal in the foreground.

This simplifies the whole process by identifying the oscilloscopes position and size only once and then using the determined information to read at the right positions at all of the section screenshots.

The reference screenshot contains no signal data. It shows the oscilloscopes as single colored plain areas. This color is specified through the input XML file and used to identify the oscilloscope.

The algorithm to determine the starting point and the size of the oscilloscopes counts the number of pixels with background color in every row and in every column. It then checks for the most appearing number for the columns and stores the index of the first and last column with this number. The same procedure is used for rows, only that for the rows there are two starting and two end points.

The points stored represent the x-coordinates and the y- coordinates of the oscilloscopes. This algorithm is functional because the oscilloscopes always got the same x-Axis start and end point and the oscilloscopes do not overlap.

In order to separate an analog oscilloscope from a digital one the color of the integrated signal is decisive.

- Analog: red = 255 and green, blue = 100 to 200
- Digital: blue = 255 and green, red = 100 to 200

It also would have been possible to identify the different types of oscilloscopes by their position. In this case the analog one would have been the upper one and the digital oscilloscope would have been the lower one. The *OsciVerifier* still distinguishes the oscilloscopes by the signal color, this makes further oscilloscope types more easily adaptable. With three oscilloscopes the order of the oscilloscope types would not be predictable any more.

<u>Identifying the actual signals:</u>

For every pixel in the oscilloscope matching the color specified in the section, the corresponding coordinates are saved (including possible error pixels) in an ArrayList. This ArrayList is later compared with the legal pixels.

5.2 Generating

In order to find any error pixels the *OsciVerifier* needs to compare the previously identified signal pixels with a list of legal pixels. This list is available in form of the measurement file.

To generate the legal pixels out of the measurement data it is necessary to:

- Scale the measurement values into pixel values. This is done by determining the ratio between pixel and measurement value (rule of proportion).
- Generate all of the pixels, drawn in between the single measurement points. The MDA itself works with different modes to connect the measurement points. In order to get equivalent results the *OsciVerifier* needs to implement certain algorithms for these modes:

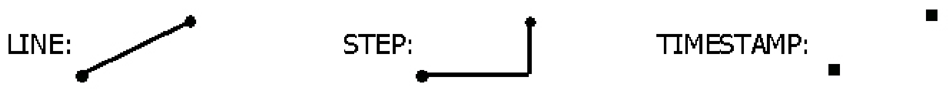


Figure 14: different drawing modes

  - <u>Line Mode:</u>

    Is implemented with the Bresenham Line Algorithm, this algorithm generates pixels which lay on a direct line between two measured pixels
  - <u>Step Mode:</u>

    Self-implemented
  - <u>Timestamp:</u>

    Not necessary for the minimum pixel size, but it needs further implementation if a greater pixel size or different marker forms (triangle, square) are requested, this was not part of the requirements.

5.3 Verifying

Verification of the correct representation of the signal pixels is achieved by comparing the signal pixels with the legal pixels including the tolerance. The tolerance which may be defined at the program startup tolerates signal pixels close to actual legal pixels. This small variance can not be avoided. It is caused by rounding errors (scaling measuring data into pixels) and by the fact that the signal line might be up to two pixels thick.

In the lower diagram we see the two red filled drawn pixels. For these two pixels the corresponding legal pixel can lay anywhere right next to one of these two pixels. With the minimum tolerance of 2 pixels (green area around the filled blue pixel) all of the possible spots for the red pixels are covered.

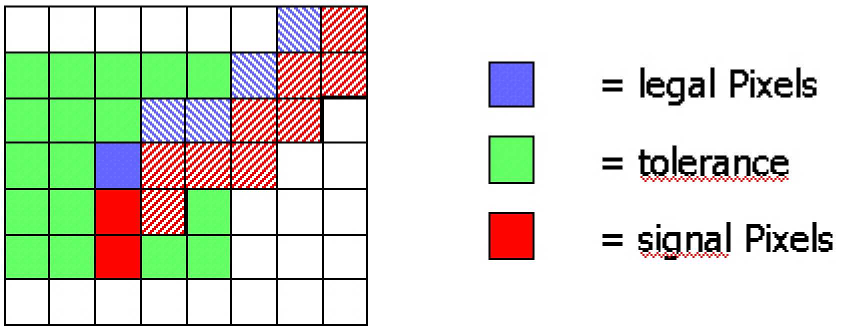


Figure 15: Example for verification and tolerance

5.4 Back tracking of failures

If an error is found its pixel coordinates get scaled back to the corresponding measurement coordinate. This is achieved by scaling. The x-value will be scaled back to a timestamp. The y-value will be scaled back to a possible measurement value.

Scaling is needed because the measurement scale does not match a pixel scale. It is done by determining the ratio between pixel and measurement value (rule of proportion).

After scaling the start x-value of the certain oscilloscope, it has to be added to the x value of the error. This is because not every oscilloscope starts with zero, but can show a range of possible values starting for example with 30 and ending with 60.

6 Conclusion

In a very interesting project, we were able to learn more about programming with Java, the field of system testing and project management. Altogether it was a great opportunity to work with and for a company like ETAS. Thanks to Mr. Gemmi we were greatly supported in all the processing of the project.

It is a pleasure to see, that students and their skills were able to solve a concrete problem of a company. Interesting to experience was that ETAS is not the only company with a problem like this. On the MediaNight, we spoke to people, having similar problems of verifying the content of pictures.

Besides, that *OsciVerifier* is already used in System test, there might be the possibility to expand or rework the tool in additional software projects or even in a diploma thesis.