Towards Model Driven Testing of Human Machine Interface Framework for In-vehicle Infotainment Platforms

Hemant Sharma, Dr. Roger Kuvedu-Libla, and Dr. A. K. Ramani

***Abstract*—Specification and implementation of tests for In-vehicle Infotainment software system is demanding and time consuming task. To reduce time and effort specification and development of tests can be done at model level. We propose platform independent test development using our extensions to xUnit Test Framework. Three phases of our test development approach: test pattern identification, test model development, and transformation are explained using components of our Infotainment Human Machine Interface Framework.**

***Index Terms*— Infotainment Human Machine Interface Framework (iHMIFw), Model Driven Architecture (MDA), Unified Modelling Language (UML), xUnit Framework.**

I. INTRODUCTION

The number of embedded software systems in automotive domain areas, such as body electronics, infotainment, and telematics applications, is steadily growing [2] [3]. Most of today's automotive software applications are developed and maintained by multiple programmers, often geographically distributed, who work on parts of the overall application code. While leading to improved code churn rates, this practice also leads to problems. For example, developers may not realize that they have inadvertently broken parts of the code.

How to test the software systems, which contains many functions in short time, and how to evaluate the quality of each software sub-system are the key problems that must be taken into consideration. Designers can no longer develop high-performance software systems from scratch but must use sophisticated system modeling method [1]. Software testing methods and objectives differ in automotive infotainment software applications from the other computer software applications. Automotive infotainment software development uses specialized compilers and development environments that rarely offer sophisticated means for testing and validation.

With the growing size and complexity of In-vehicle infotainment HMI software applications, software verification and validation techniques such as testing and model checking are increasingly important. While testing focuses on the actual behavior of the program, model checking focuses on its business and logical interaction model. Also for infotainment applications, Testing and model checking are complementary: testing is lightweight but incomplete while model checking is heavyweight but complete.

User interfaces built for In-vehicle infotainment applications traditionally presented a challenge to development testing because of the following factors:

- The complex nature of the underlying graphics framework.
- The coupling of presentation and business logic within a User Interface.
- The lack of support from underlying architecture and intuitive automated testing frameworks.

Of course, the first two factors are nothing new - graphical frameworks are complex by nature and adding business functionality to a HMI application has always posed a barrier to testing. On the other hand, a number of handy frameworks have popped up over the last few years that actually facilitate testing of HMIs of software systems.

Our goal in this paper is to present our ongoing efforts towards the specification and efficient execution of testing of components of infotainment HMI applications by means of the model driven approach, which includes:

- The clarification of the methodological approach for the introduction of model-driven testing in the Infotainment HMI context.
- The identification of a testing metamodel able to be used in practice in the context of In-vehicle Infotainment HMI applications, and close to the xUnit testing frameworks [21, 22].

Hemant Sharma is Software Engineer at Delphi Delco Electronics Europe GmbH, Bad Salzdetfurth, Germany.
(e-mail: hemant.sharma @ delphi.com).

Dr. Roger Kuvedu-Libla is EMC-Competency-Leader at Delphi Delco Electronics Europe GmbH, Bad Salzdetfurth, Germany.
(e-mail: roger.kuvedu.libla @ delphi.com).

Dr. A. K. Ramani is Professor at School of Computer Science, Devi Ahilya University, Indore, INDIA. (e-mail: headscs@dauniv.ac.in).

- The selection of test modelling mechanisms capable to represent the common and the variable parts in the tests architecture, in usable and efficient way.
- The identification of derivation mechanisms that cover the transformation of test specifications into platform specific test cases.

This paper is organized as follows: In the following section an overview of related research is provided. Section 3 shortly explains the architecture of iHMIFw. In section 4, we describe the test framework architecture. Section 5 describes the model driven test development approach. In section 6, we elaborate the future activities and finally conclude the paper.

II. Background

The testing techniques for automotive infotainment software systems are based on the specification of a program. This specification driven testing is also called black-box testing. White-box testing on the other hand is based on knowledge on the implementation of a program [13]. Quite some confusion exists between these definitions [14]. Unit testing can be seen as a mix between both techniques, hence called grey-box testing. Developers of unit tests can use some specific implementation knowledge for writing tests. On the other hand, unit tests can also be written according to a (detailed) specification.

The rising popularity of automated unit testing seems to have inspired the creation of several GUI test toolkit projects. JFCUnit [19] has a tool class called *JFCTestHelper* for examining the state of the graphical environment, as well as massaging the event stream to programmatically manipulate components. Tests are coordinated with *JUnit*. *Jemmy* [20] is a library for automating Java GUI applications. It has an advanced abstraction tree for finding, examining and manipulating specific graphical components.

In the field of software development the complexity of systems has been reduced by using abstract specifications. Models are used to represent the more complex entities to understand, communicate, explore alternatives, simulate, emulate, calibrate, evaluate and validate software [15]. Therefore it is a logical consequence to represent test code as models, too. In the case of test models all advantages mentioned before are provided. The benefit of models lies in their abstractness as opposed to implementation specific concreteness of code [16]. Model driven test development does not oblige which software development methodology has to be used. It suits to testing first methodologies like Agile [17] and Lean [18] development as well as for Model Driven Development.

Most approaches to model-based testing [5, 6, 7] used in automotive infotainment domain do not consider the separation described in MDA, i.e., they are either tailored towards a specific target platform or they are generic in this respect, taking into account platform-independent model information only.

In order to benefit from the separation of PIMs and PSMs in the generation and execution of tests, the strategy of Model-Driven Testing [16] has to refine the three classic tasks of model-based testing:

- the generation of test cases from models according to a given coverage criterion,
- the generation of a test oracle to determine the expected results of a test,
- the execution of tests in test environments, possibly also generated from models.

With an appropriate level of detail in the interface specifications of domain-specific components, it is possible to automatically generate some or all of their test cases. In fact, test cases can be specified and generated in parallel to the specification and generation of components.

A lightweight automation framework for system tests can extend the benefits of extreme programming (XP) unit testing [8] to a higher level, supporting test first development for system tests, and decreasing the difficulty of writing system tests.

III. Infotainment HMI Framework Architecture

This section provides an overview of architecture of our HMI framework, iHMIFw, for In-vehicle infotainment applications. The framework has been developed using MDA methodology and partitioned into independent components based on functional areas. Figure 1 partially represents the components of the HMI framework.

iHMIFw emphasizes on application development using the iterative practice of Test Driven Development [4], to have the following benefits:

- Shorter code implementation time
- Reduction in code defects
- Fewer instances of overcomplicated and unnecessary code
- Reduced likelihood of introducing bugs when fixing bugs, refactoring or introducing new features.

The components of HMI framework are organized to make the framework scalable and flexible. The framework has a set of core components and some optional components.

Core components provide the bare minimum functionality that is required for an HMI application. Optional component can be configured along with framework to provide additional functional interfaces. These core components help to design HMI for an Infotainment application, independent of the application, with basic features such as Widgets, Views etc.

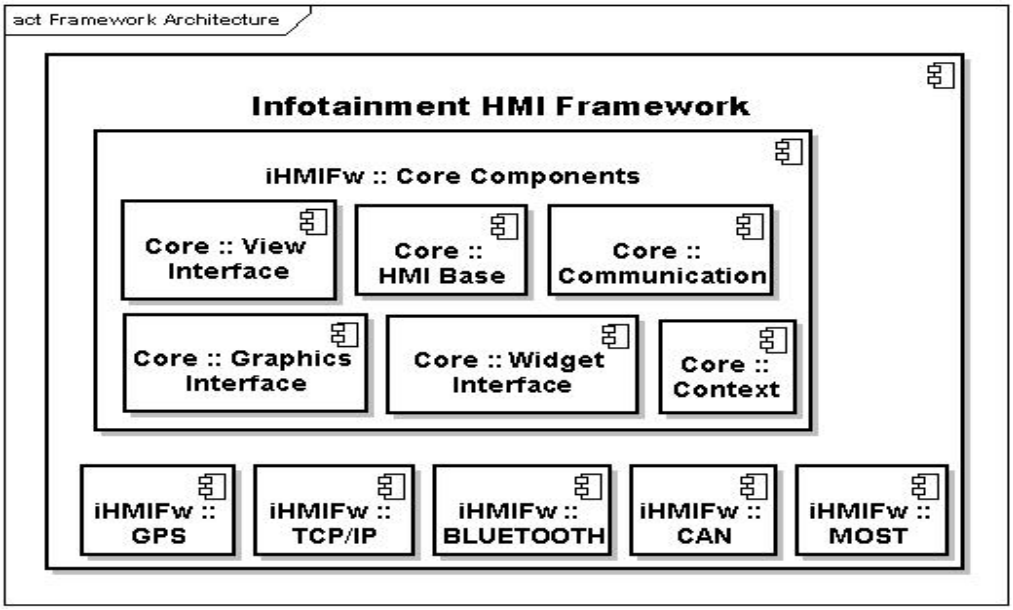

Figure 1: iHMIFw Components.

Responsibility of individual core component is described below.

Core :: View

This component is responsible for HMI View creation. It provides interfaces to describe the appearance and behavior of the view. Further, this component is also responsible to define the view tree structure for an HMI application and defining the view state transition.

Core :: HMIBase

HMIBase component defines the structure of HMI applications. This component enables the HMI application to interface with platform specific aspects such as startup and shutdown of application.

Core :: Communication

The communication between components of HMI Application as well as communication with external applications is supported by this component.

Core :: Graphics Interface

This component interfaces to graphics resources in the system. This includes graphics libraries, display access handlers, image management and font resource files.

Core :: Widget Interface

HMI Applications are allowed to use specific widgets available in infotainment application specific widget libraries. This component provide interface to the widget resources, the HMI application intend to use.

Core :: Context

This component provides interfaces that help HMI Applications to define the context for its individual views. Further, this component has mechanism to read context configuration information from a XML file.

IV. Test Framework Architecture

The test framework we propose here is inspired by 'xUnit Testing Framework'. Over the years the xUnit testing framework has become a de facto standard. Most integrated development environments have xUnit integration via a plug-in. The framework is referred to in many programming, software development, maintenance, reengineering as well as software testing books [21, 22, 23]. The xUnit framework is being adopted by industry as well, by means of JUnit itself or its commercial derivatives.

In the subsections below, we describe the derived xUnit patterns and the meta-model for the test framework.

A. Patterns for Model Testing

In this section, we provide an overview of test patterns derived from xUnit patterns to enrich behavioral, logical and organizational test development for Infotainment HMI components.

HMI Mock Pattern

A common problem with HMI unit tests is to test complex objects that rely on external systems. A unit test must make some assumptions on the state of the object to test. This state should be restored on each test run. External systems do not always allow this. *Mock objects* are a technique to prevent

this problem. Instead of using the actual implementation, that possibly uses external systems, a fake - hence the name mock - object is used. A mock object contains no concrete implementation and is solely used to validate whether the application code behaves as expected.

Figure 2 shows the *Mock Test* pattern that has been customized for iHMIFw. Classes *MockView* and *MockService* implement the same interface as a real View and Service classes respectively. Objects of *MockView* class shall serve as observation point for individual view-behavior verification.

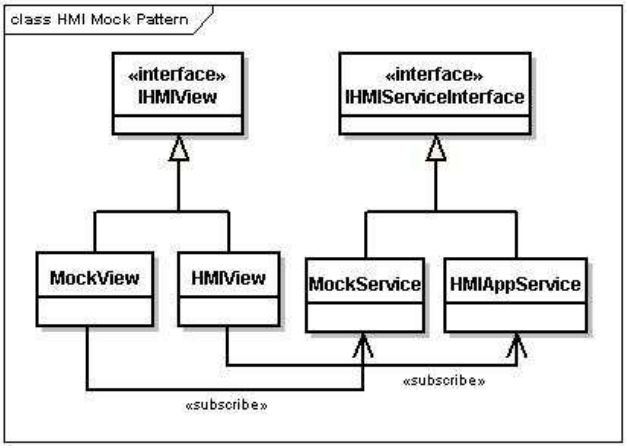

Figure 2: Mock Test Pattern for iHMIFw.

Similarly, the objects of class *MockServic*e shall be useful for communication state-behavior verification. Both classes shall produce 'lenient' mock object instances such that the tests are independent of the order of their execution.

HMI Layer Test Pattern

The Infotainment HMI applications based on iHMIFw inherit the Layered Architecture from the framework to separate major technical concerns. Most applications shall have, at minimum, some kind of presentation (user interface) layer, a business logic layer or domain layer and a persistence layer. Some future layered HMI application architectures may have even more layers. It is difficult to get good test coverage when HMI application is organized as integration of component in layered fashion. Such application architecture forces the use of 'Indirect Testing' of individual function group of components.

In order to get good test coverage of logic of each layer these application shall be supported by iHMIFw specific Layer test pattern. This iHMIFw specific Layer Test pattern has been derived from xUnit Layer Test pattern [6].

Figure 3 presents the custom Layered Test pattern for iHMIFw components and applications based on iHMIFw. Layered Test interfaces are either realized or extended to derive layer specific test. *ViewLayerTest* and *ServiceLayerTest* are the components that collectively represent interface test classes for View and Service Layers of iHMIFw respectively. *View Layer Test* class shall provide objects to test the presentation logic of Views independent of the business logic of HMI application. Service communication logic test objects shall be provided by *Service Layer Test* classes.

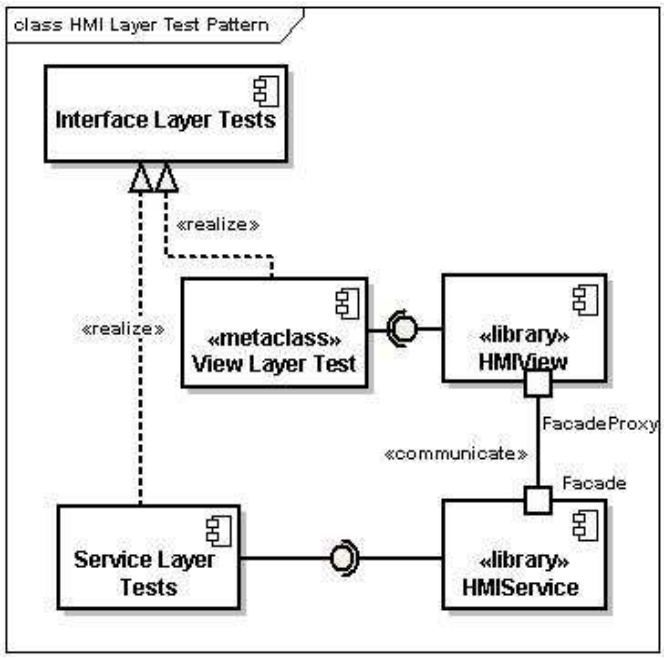

Figure 3: Layered Test pattern for iHMIFw Components.

HMI Test Organization Pattern

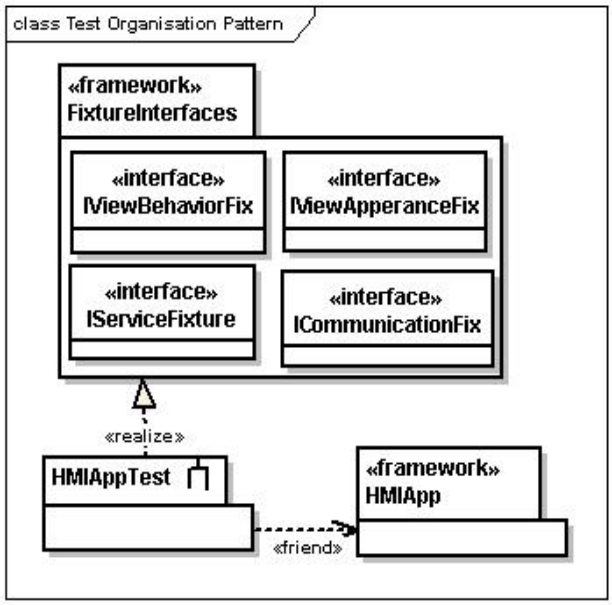

Figure 4: Test Organizational Pattern for iHMIFw.

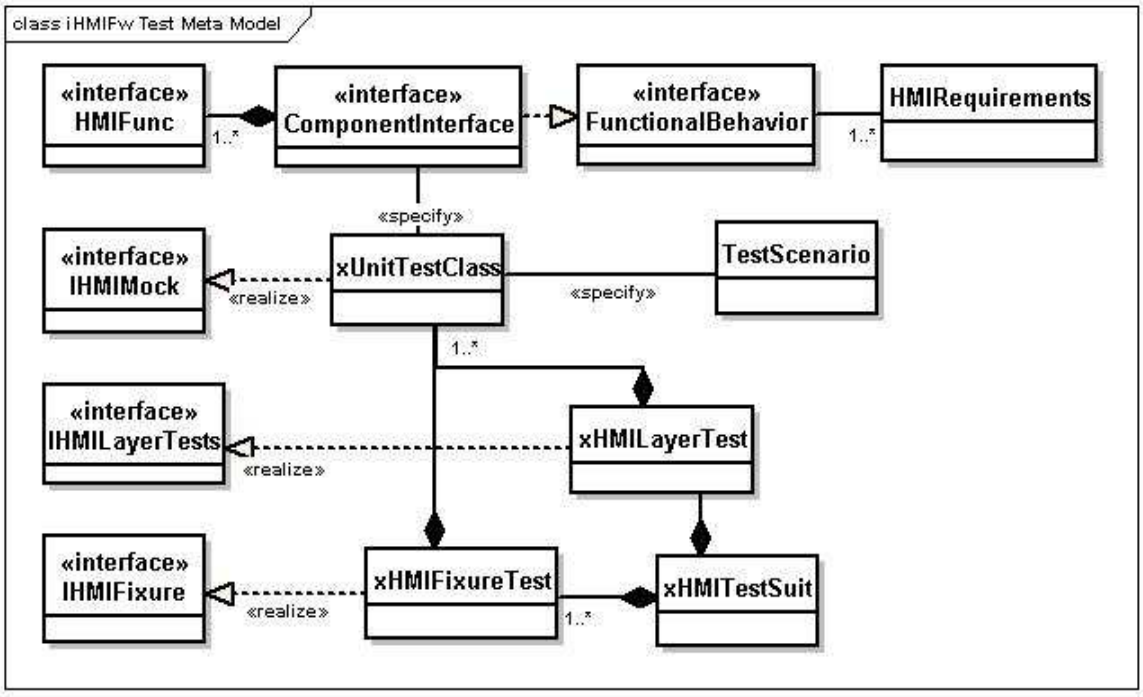

Figure 5: iHMIFw Test Meta-model

In-vehicle Infotainment HMI applications based on iHMIFw are expected to have significantly large number of structural and behavior test classes.

Further, as the number of test methods shall grow, an efficient mechanism is required to decide wrapper test classes to hold the test methods. To get a simplified structure of application tests, the iHMIFw provides a *Test Organization* pattern derived from the *Test Fixture* pattern defined in [6].

Figure 4 presents the *Test Fixture* interface packages that establish the *Test Organizational Pattern* for iHMIFw test organization. Component specific partition of *Test Organization* interfaces shall help to achieve a declarative style of Test Suit development.

B. Test Meta-Model

The specification of a meta-model, capturing entities and relationships of interest during *xUnit* based testing, is facilitated through the consistent terminology in testing literature and supporting tools.

The metamodel for testing presented in the figure 5 is a conceptual framework able to cope with variability in testing [9], therefore, although it follows the general principles stated by the UML profile for testing [10], modifies and extends the profile in several aspects. In particular, our metamodel includes some elements already available in the profile, such as *Test Class, Test Suite*, and the concept of *Test Scenario*. The mapping of the *Test Class* to Component Interface as well as the realization of extended Test Pattern interface in conjunction to 'Requirements Model' testing [9], are however, unique in our approach.

The metamodel shows the main entities of the test approach and their relationships. The HMI requirements are the basis of the test; in particular, requirements (both generic in the Infotainment HMI family and specific to a subset of Infotainment functionality in the integrated system) drive the design of test classes. The *xUnitTestClass* is a group of test cases related with certain HMI functionality. These test cases are based on *Test Scenarios*, which are derived from a model of the HMI behavior specification.

V. Test Specification and Transformation Directives

To accent the agile part of our model-driven test development approach we want to support the test-driven development by enabling model-driven unit, layer and integration test development. Therefore, beside the generation of test case model from application model we want to automatically generate platform specific test cases from our test model.

A. Approach Towards Test Development

In this section we present the steps of our approach for test development. Activity diagram in figure 6 shows our model driven test development process enabling model driven unit and integration testing.

For the iHMIFw xTest patterns and meta model has been developed, as described in previous sections, and stored in a repository. The test development process can be summarized in the following steps:

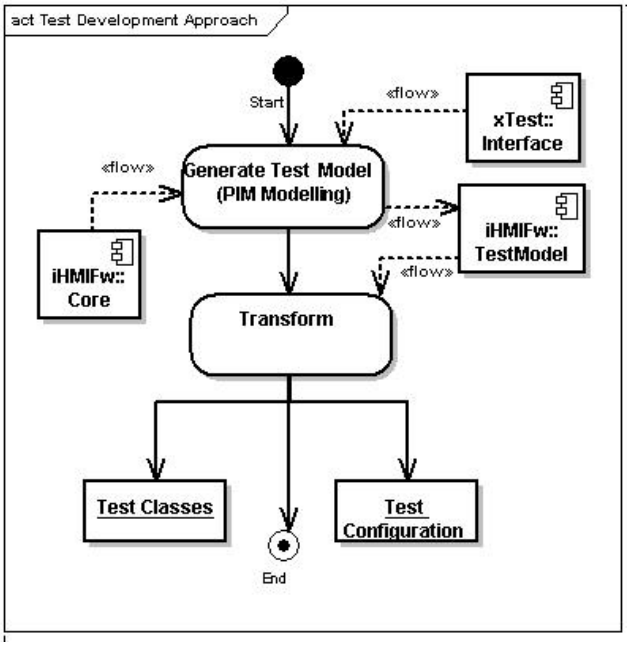

Figure 6: Test Development Approach

- *Generate Test UML Model* - First step in Test Development is to develop test model from the UML design model. Class diagrams from the design model serve as basis for test class generation. Interfaces from xTest::Interface are extended to get the test classes for HMI classes in class diagram from design model.
- *Test Model Transformation* – Test classes and the relationship between them, as represented by test class diagram, is transformed to provide source code and test configuration .

B. Platform Independent Test Modeling

The basic idea of our test development approach is the specification of test classes by using defined xUnit test interface classes. Further, the test classes are enriched with pre- and post-conditions (Infotainment Contract), which can be viewed as a test oracle [11, 12] and runtime assertion checking can be used as a decision procedure for a test oracle.

In order to keep the test model simple for presentation purpose, we have partially used the core components of iHMIFw in class diagram of Figure 7 below. This figure represents partial model for the framework and form the basis to develop platform independent test model.

Interface class *IObject* is the base class from which all the framework classes have been derived. *IWidgetBaseInterface* provides the base structure for Widgets that shall be used in views of HMI applications. HMI Views shall use *ViewBase* to inherit minimal functionality for the view. The views shall contain multiple widgets to manage the appearance and collective behavior. *WidgetAppearance* and *WidgetBehavior* classes provide the basic appearance and behavior features respectively.

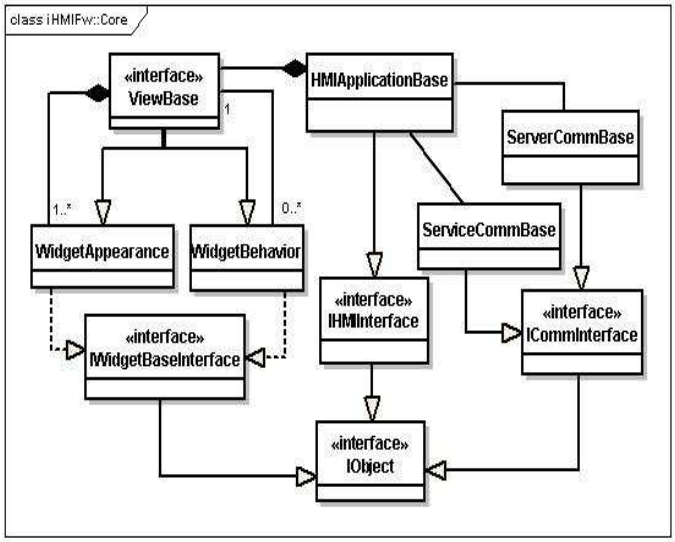

Figure 7: iHMIFw::Core PIM

Figure 8 shows the Platform Independent Test Model (TPIM) derived from the platform independent model for core components of iHMIFw. Corresponding to every class in PIM model of the framework, a test class has been shown in TPIM. IHMIMock represents the behavioral test interface (Mock object interface).

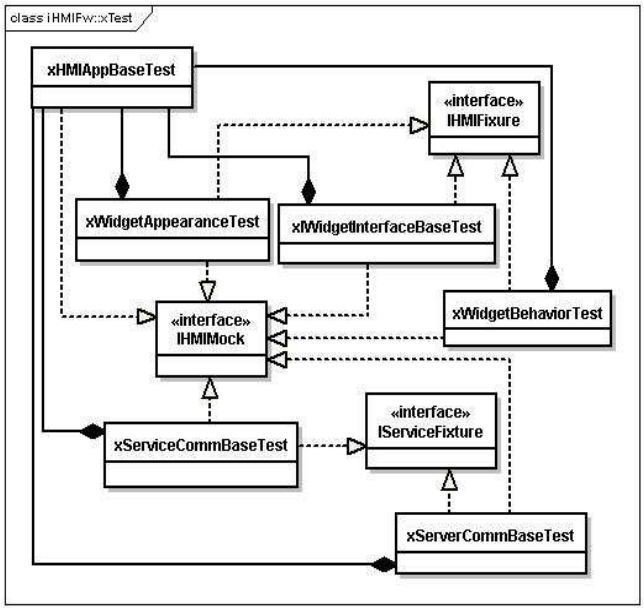

Figure 8: iHMIFw::Core Platform Independent Test Model

IHMIFixture and *IServiceFixture* bring in the test suit organization strategy for test classes. The test classes realizing these interfaces provide objects that individually represent Mock test cases. Defined collection of objects from test class shall form scenarios for layer tests. A fixture configuration along with test classes shall organize tests into a test suit. Test collection definition and fixture configuration

are the topics for our future work.

C. Platform Specific Test Model

Similar to classical unit-testing, our test items are operations. The behaviour of an operation is dependent of the input parameters and the system state. Thus, a test case has to consider the parameter values of an operation and a concrete system state.

A test case for an operation consists of concrete parameter values and a concrete system state. We can generate a test case for an operation from our model in three successive steps.

- In the first step, we generate values for the input parameters of an operation as specified in the class diagram.
- we initialize the pre-condition of an infotainment contract with the parameter values generated in step one. The variables in the parameter-list are used to restrict the attribute values of objects in the pre-condition.
- In the last step of our test case generation, we have to find out how to generate a system state which contains the object structure found in step two.

```
/**
  *  generated - xWidgetBaseInterfaceTest
  */
class xWidgetBaseInterfaceTest:
   public IHMIMock, public IHMIFixture
{


protected:
   // Test case setup
   void TestSetUp()
   {
      // Initialize mock test parameters
      _initWidgetBaseMock();

      // Initialize fixture test parameters
      _initWidgetBaseFixture();
   }

public:
   // Test - Widget creation
   void testWidgetBaseCreate(int widget_Id )
   {
      WidgetBase  testWidget(widget_Id);
      testWidget.initialize();
   }

   // Test Widget - paint
   void testWidgetBasePaint()
   {
      WidgetBase  testWidget;
      testWidget.initialize();
      testWidget.paint();
   }

}
```

Figure 9(a) : PSM - iHMIFw::Core Generated Test Case

Figure 9(a) shows, out of the test specification model a platform independent test code that has been generated for the class *xWidgetBaseInterfaceTest*. The class inherits Mock interface for behavioral tests for the widgets. This leads to incorporation of one test method corresponding to each method of *WidgetBaseInterface* class. Secondly, inheritance of fixture *interface* helps in partitioning of test methods for complex widgets. Widget Test objects shall be initialized at test startup via configurable test data.

All test cases need access to representative data to use in testing the functionality of the HMI application component. Test PIM transformation, in second step, generates XML configuration information corresponding to test classes. This configuration information mainly describes test data and test execution setup sequence. . Test objects shall use data out of XML configuration files for two main purposes:

- for initialization parameters for test execution ,
- for determining the sequence of execution for test methods.

Figure 9(b) represents an example of XML configuration file for *xWidgetBaseTest* test class.

```
<?xml version="1.0" encoding="utf-8"?>
<root
xmlns:test="iHMIFw/core/widget/xWidgetBaseTest">
  - <test:case test="setUp" result="pass"
                      desc="test:case  setUP">
    -  <testingSetUp nested="false"
                       attrib="widget_id">
      <test:attribute id="w0x12890" name="MenuGear"
                       result="fail"/>
      <test:attributeRef>b:panel</attributeRef>
      </test:attribute>
      <child child="method" />
      - <test:case test="createWidget" result="pass"
                    desc="test:case createWidget">
        - <testingBody nested="true">
          <child child="method" />
        </testingBody>
      </test:case>
    </testingSetUp>
</test:case>
```

Figure 9(b) : PSM - iHMIFw::Core Test Case Configuration

The transformation will result in source code and configuration XML repository for set of test classes corresponding to iHMIFw::Core components. The test library resulting from these test classes shall be usable for execution from test automation tools [24, 25] that are using *xUnit Test Framework*.

VI. Conclusion

In this paper, we proposed an approach for test case development for components of HMI framework for In-vehicle infotainment platform implementing model driven approach. By using extended xUnit test patterns, it is possible to derive test model for iHMIFw components. We have established test patterns, by extending the existing xUnit patterns that will help in platform independent test modeling. Further, we presented meta-model for development of iHMIFw test PSM.

We have shown that the UML design model of iHMIFw can be used to develop platform independent test model enriched with unit, coverage and integration test classes. Further, it has also been shown, how transformation of test PSM will provide test classes along with test configuration.

The growing complexity of HMI software applications in In-vehicle Infotainment systems needs solutions which allow complexity reduction by raising the abstraction level. Model driven test development is a step further to achieve more manageable and transparent HMI development. We have shown that model driven test development can be adapted for unit testing, integration testing, system testing and performance testing of HMI components of infotainment platform. By using UML, we build on a well known standard that is predominantly used in today's model-driven development processes. Further, we presented how to use the test patterns in a model driven test development process.

In future work we will have to concretize our model-driven unit testing approach. At First step, we shall refine the extended patterns and integrate them in to iHMIFw as a component. Secondly, we shall investigate for dedicated test automation framework. Finally, we shall establish a test tool chain for automated testing for iHMIFw and components of HMI application based on it.

References

[1] D. Gajshi et al., "Specification and Design of Embedded Systems", Prentice Hall, Englewood Cliffs, N. J., 1994

[2] Ren Yu, Wan Jian, "Software Simulator of Embedded Application System", Computer Application, Vol. 11, No. 7, July. 2004, P.144-146.

[3] Ren Yu, "Threads Communication Performance of Embedded Simulator", Computer Application, Vol. 25, No. 25, Dec. 2005, P.12-14.

[4] David Astels, Test Driven Development: A Practical Guide, Upper Saddle River, NJ: Prentice Hall PTR, 2003.

[5] Basanieri, F., A. Bertolino and E. Marchetti, A UML-based approach to system testing, in: J.-M. Jezequel, H. Hussmann and S. Cook, editors, UML 2002, LNCS 2460 (2002).

[6] Briand, L. and Y. Labiche, A UML-based approach to system testing, in: M. Gogolla and C. Kobry, editors, UML 2001, LNCS 2185 (2001).

[7] Martena, V., A. Orso, and M. Pezze, Interclass testing of object oriented software, in: Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), 2002.

[8] XUnit Test Patterns. http://xunitpatterns.com/

[9] Coplien, J., Hoffman, D., Weiss, D.: Commonality and Variability in Software Engineering. IEEE Software, November 1998.

[10] OMG, UML Testing Profile (final submission) by Ericsson, IBM, Fokus, Motorola, Rational, Softeam, Telelogic. March 2003.

[11] Antoy, S., Hamlet, D.: Automatically checking an implementation against its formal specification. IEEE Transactions on Software Engineering 26(1) (2000) 55–69

[12] Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Transactions on Software Engineering 24(3) (1998) 161–173

[13] Bret Pettichord. Thinking outside the boxes, 2004.

[14] Alexandre Petrenko and Andreas Ulrich, editors. Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003, volume 2931.

[15] Thomas, D.: Programming with Models - Modeling with Code. The Role of Models in Software Development. in Journal of Object Technology vol .5, no. 8 (November -December 2006) pp. 15–19.

[16] Stahl, T., V¨olter, M.: Modellgetriebene Softwareentwicklung. dpunkt- Verl. (2005) of Lecture Notes in Computer Science. Springer, 2004.

[17] Beck, K.: Extreme Programming Das Manifest. Addison-Wesley (2003).

[18] Poppendieck, M., Poppendieck, T.: Lean Software Development Number ISBN 0-321-15078-3 in The Agile Software Developement Series. Addison-Wesley (2003)

[19] JFCUnit. http://jfcunit.sourceforge.net/

[20] Jemmy. http://jemmy.netbeans.org/

[21] A. Hunt and D. Thomas. Pragmatic Unit Testing in C# with NUnit. The Pragmatic Programmers, 2004.

[22] J. Langr. Agile Java Crafting Code with Test-Driven Development, chapter Lesson 15: Assertions and Annotations. Prentice Hall, 2005.

[23] M. Feathers. Working Effectively with Legacy Code. Prentice Hall, 2005.

[24] StrutsTestCase for JUnit. http://strutstestcase.sourceforge.net/

[25] JUnit Test Decorators. http://www.clarkware.com/software/JUnitPerf.html