2007-01-0780

Creating Human Machine Interface (HMI) Based Tests within Model-Based Design

Chris Fillyaw, Jonathan Friedman, and Sameer M. Prabhu
The MathWorks

ABSTRACT

Many of the multimedia and convenience features in today's passenger vehicles involve Human Machine Interfaces (HMIs), such as the radio face plate or the remote key fob. The functional requirements for these systems are often written in terms of the customer interaction with the interface device. In the past, design engineers would not begin to test requirements for these systems until prototype hardware was available. However, many product development organizations are shifting from this hardware-based traditional development cycle, which relies on designing via a prototype and test iteration, to Model-Based Design. Unfortunately, testing systems with complex human machine interface requirements becomes less intuitive when the prototypes are removed from the design process, because the test cases must be scripted into the modeling environment instead of being applied directly to a prototype of the interface device. In this paper we will show how engineers can create a "soft" representation of an automotive HMI and record the test procedure when specified as a series of interactions with the interface, such as button presses. Next, we will demonstrate how the test procedure can be captured and exported to an editable file for re-use with Model-Based Design. Lastly, we will show how a test file can be used to populate a test harness within the Simulink® software environment.

INTRODUCTION

The need to bring innovative, high-quality automobiles to market faster and at the same time comply with stringent regulatory requirements is driving the increased use of models during the design and realization process. The ability to model and simulate various automotive systems prior to building hardware enhances the product development process by allowing manufacturers to test whether the system meets requirements using virtual rather than physical prototypes. By using such virtual prototypes, in the form of models, designers can explore multiple design alternatives to optimize the design and discover errors in the system early on. Thus, Model-Based Design provides efficiencies in product development that enable companies to deliver products on time, to remain within budget, and to fulfill initial requirements.[1] The latest Model-Based Design tools can also generate prototype and production code from a model automatically, significantly decreasing development time. As a result, Model-Based Design is an integral part of the automotive systems development process.

In this paper we explore the application of Model-Based Design for the development of multimedia and convenience features in today's passenger vehicles. The MATLAB® and Simulink® software environments are used throughout the design process for developing and testing these systems, because they provide high-level formalisms that allow the modeling of such systems and also allows automatic generation of code to test and implement these systems. [2] [3]

In this paper we discuss in detail the application of these tools to the development of a specification of a radio face plate as an example of typical multimedia system in a car. We also illustrate the testing of the system to ensure that the design meets requirements and to identify inconsistent requirements. The paper is organized as follows: Section 2 describes in detail the problem framework that is used throughout this paper, and discusses Model-Based Design concepts and how they can be applied to the same problem. To conduct a detailed and realistic testing of the radio face plate specification it is necessary to build a Human Machine Interface (HMI) to the specification. Section 3 focuses on the issues involved with building such a HMI. Section 4 then deals with using this HMI to test the specification and also discuses creating test sequences based on the HMI interactions, which can then be reused not only for testing the specification but also the final implementation. The conclusions are drawn in Section 5.

ROLE OF EXECUTABLE SPECIFICATIONS IN THE DEVELOPMENT OF MULTIMEDIA AND CONVENIENCE FEATURES

It is estimated that electronics and software content will make up 40% of a vehicle's cost by 2010.[4] In the early days, electronics were used primarily to ensure that vehicles met stringent emissions and fuel economy requirements by replacing conventional mechanical systems with electronic systems. Currently, in addition to the focus on emissions and fuel economy, automotive manufacturers use electronics to introduce advanced multimedia and convenience features to attract technology-savvy buyers. These features focus on assisting the driver through technologies such as hands-free phone operation, using the car's radio and Bluetooth networking, and also with providing relevant information to the driver such as upcoming traffic information and the need to change the planned route to the destination. In addition to assisting the driver, these systems also entertain passengers through on-board systems such as satellite radio, DVD players, and so on, which can be accessed through a common interface. Automotive manufacturers see such systems as a key source for differentiating themselves from the competition, and also as a lucrative revenue stream. As such there is an increased emphasis on developing and deploying these systems to meet consumers' varying requirements, while being simple enough to operate and use, and of high enough quality to avoid the need for costly recalls and software fixes.

ISSUES IN MULTIMEDIA AND CONVENIENCE FEATURES DEVELOPMENT

A key aspect of the multimedia and convenience features is that they have to be easy to use. Thus a significant amount of time and effort is devoted to designing the HMIs for these systems in addition to designing the underlying electronics. An example of such a system is the radio face plate or the remote key fob commonly found in automobiles today. The starting point for the design of these systems is usually their functional requirements, which are often written in terms of the customer interaction with the interface device. For example, it is common to have requirements for the radio face plate such as:

Requirement 1: When any radio preset button is depressed for less than three seconds, the radio shall tune the radio receiver to the station value stored in the radio preset.

Requirement 2: When any radio preset button is depressed for three seconds or more, the value of the preset shall be set to the current station to which the radio receiver is tuned.

A typical development process takes these requirements and proceeds through the system design and implementation stages to test the system to determine if it meets the requirements or not. If it does not meet the requirements, an expensive and time consuming debugging process follows to determine if the implementation is faulty or if one or more requirements are inconsistent with each other. In either case, because functional and performance testing does not begin until prototype hardware is available, there is a significant time lag to turn around design changes to meet requirements changes or fix design errors. To address these issues, many product development organizations are shifting from this hardware-based traditional development cycle, which relies on designing with a prototype and test iteration, to Model-Based Design.

INTRODUCTION TO MODEL-BASED DESIGN

Model-Based Design lets development organizations address the challenges of increasing product complexity, more stringent performance requirements, and shorter product development cycles. By using models in the early design stages, engineers can create what are known as "executable specifications" that enable them to immediately validate and verify specifications against the requirements. Validation ensures that the requirements are correct and that they represent the intended behavior. Verification ensures that the outputs of each step satisfy the step's inputs (i.e., the system satisfies its requirements). Thus, Model-Based Design allows engineers to detect errors earlier in the development process when the cost to fix them is less expensive. Further down the design process, models can be used to communicate between engineering teams with different specializations, allowing them to work together and to communicate between stages in the overall process. Moreover, initial design models can be incrementally extended to include increasing implementation detail. Thus, Model-Based Design allows engineers to explore different design alternatives early in the design process using the models which are part of the executable specification. After the concept phase is completed, the same executable specification (containing the model) is used by the next team which may need to include detailed implementation effects into the model. This process continues with each part of the design team elaborating the same model used by the previous team and testing their contribution to the design. In contrast, a document-centered approach requires each design stage to generate new artifacts or design documents to communicate the state of the design as it passes from one stage to the next.

In addition to elaborating the model at each design step to minimize the non-value added work and possible introduction of errors, many companies are using the models for multiple purposes. For example, the same models used to communicate the design for software or hardware design can be used for hardware-in-the-loop testing. , Moreover, these activities can be automated so as to prevent introduction of errors that can occur during manual implementation and realization tasks, and further shortens the path to product delivery by generating code for testing, calibration, and the final production. An important benefit of Model-Based Design is the traceability of design decisions all the way down to the implementation, so test results can be directly interpreted as high-level design decisions. Finally, even though electronic models are easier to navigate than paper documents, the formal system design process still requires detailed documentation. Advanced tools allow automatic generation of this documentation from a model, while Model-Based Design forces the design process as well as the final product to be documented for maintenance and future developments. The next section discusses how Model-Based Design can be used for developing a radio face plate system.

MODEL-BASED DESIGN FOR RADIO FACE PLATE DEVELOPMENT

MATLAB and Simulink are used throughout the design process because they provide high-level formalisms such as Stateflow® [5] to support detailed modeling of both the radio face plate specification and the testing infrastructure required to test the face plate. A radio face plate is typically designed to meet requirements, some of which are mentioned below:

Power is turned on: When the battery is powered (key in accessory mode or key on), the audio unit goes into the 'PowerOn' mode. The system enters the 'Standby' state.

System is turned on: When the user turns the system on, the system moves into the 'On' state.

These requirements are typically captured in paper form. Traditionally, to verify if the embedded controller drives the system into the standby state when the ignition state is key-on, we need to wait for prototype hardware. Within Model-Based Design, , we can specify these requirements using Stateflow, which not only provides us a visual interpretation of the system functionality but also allows us to test the functionality virtually. Stateflow is used to elegantly model the discrete event algorithm by exploiting the hierarchical state behavior of Stateflow (for example, the Power On state has two sub-states, viz., Standby and On; also On itself may have sub-states), as shown in Figure 1.

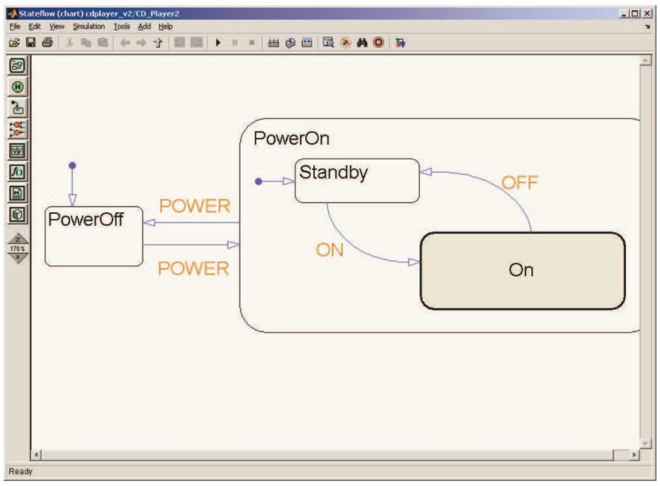
Stateflow (chart) cdplayer_v2/CD_Player2
File Edit View Simulation Tools Add Help
PowerOn
Standby
OFF
POWER
PowerOff
ON
POWER
On
Ready

Figure 1: Stateflow® representation of radio face plate logic.

The Stateflow model then becomes the heart of the executable specification of the system. Since this specification is executable, unlike traditional paper specifications, the behavior of the system can be verified by simulating the model Simulation is achieved by submitting the model to a variety of inputs that correspond to user commands, to verify that the requirements are correctly captured in the Stateflow diagram. Inputs can be created using the Signal Builder functionality in Simulink, as shown in Figure 2. The animation capability of Stateflow provides visual feedback regarding the correct functionality of the algorithm.

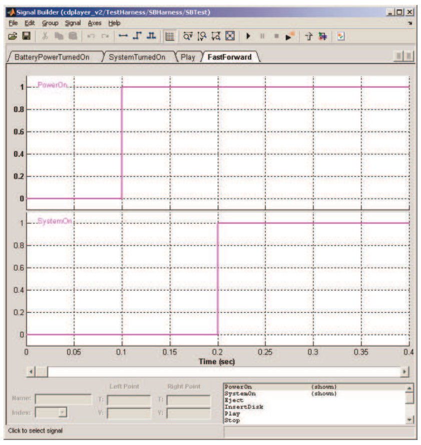

Figure 2: Test case representing user commands.

THE NEED FOR A HUMAN MACHINE INTERFACE (HMI) IN AN EXECUTABLE SPECIFICATION

Executable specifications often specify the desired behavior of a system to external inputs, as described above. For example, in a “throttle by wire system,” the driver’s input to the algorithm is via the throttle pedal. The executable specification then specifies the desired response of the system to this input. In this case, there is only a single user input to the system. In general, these types of systems with a small number of human inputs can be captured using a time-based trace, or modeled using common constructs such as steps and ramps. However, there is a class of systems in automotive engineering that can have multiple inputs or a series of multiple inputs from the driver or passengers over time. In these cases, it is neither practical to try to model the inputs, nor to modify the hardware to create a trace of the user inputs. Instead, it makes sense to create a graphical representation of the HMI hardware to be included with the executable specification. Through the inclusion of the HMI, engineers can interact with the specified algorithm using the same form of input as the end user.

For example, many of the multimedia systems in today's passenger vehicles involve sophisticated HMIs that require a series of inputs from the user to achieve their desired result, e.g., programming the radio frequency preset buttons or using a navigation system. The functional requirements for these systems are often written in terms of the customer interaction with the interface device. For example, it is common to have requirements such as:

1. When any radio preset button is depressed for less than 3 seconds, the radio will tune the radio receiver to the station value stored in the radio preset.

2. When any radio preset button is depressed for 3 seconds or more, the value of the preset will be set to the current station to which the radio receiver is tuned.

Traditionally, such requirements are tested using prototype hardware. However, since executable specifications allow for testing prior to hardware availability, engineers now must develop test scripts to exercise the algorithm. Unfortunately, since there are hundreds of these series of HMI inputs required to fully test a typical radio, modeling the inputs as a time-based test vector can be time consuming and monotonous. To address this issue, engineers add graphical or "soft" representations of the proposed HMI to the executable specification to allow design and test engineers to interact with the algorithm.

The steps of the test procedure can then be recorded and edited (if necessary) to be used later to automate the testing process for future design iterations within Model-Based Design.

USING HMI'S TO CREATE TEST VECTORS

To demonstrate the capability of using an HMI to create test vectors, an automotive radio system will be used throughout this example. The radio is controlled through a face plate HMI containing a number of buttons and a display. This example focuses on developing the "soft" HMI representation and using it to create test vectors examining the following topics:

1. Develop the "Soft" HMI representation.

2. Capture the user inputs to the HMI and populate a Signal Builder block with the test vector.

3. Exercise the model with the test vectors, capture the system response, and populate the Signal Builder block to form one test case with acceptance criteria.

4. Create test vectors based on requirements specifications.

Figure 3 displays the "soft" HMI representation that will be constructed in this example providing the capability to create test vectors.

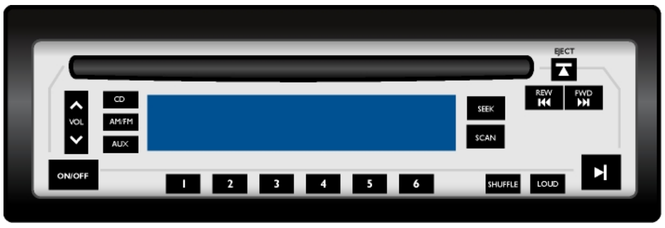

Figure 3: Radio face plate HMI.

The CD player system model shown in Figure 4 is the model under test. On the far left, the "TestHarness" subsystem contains a Simulink Signal Builder block and on the right, the "Verification" subsystem contains the verification blocks. The role of the Signal Builder and verification blocks should be evident after following the example presented in this paper. Throughout this example, the contents of the Signal Builder block will be generated from the button presses on the HMI and through exercising the model to capture acceptance criteria.

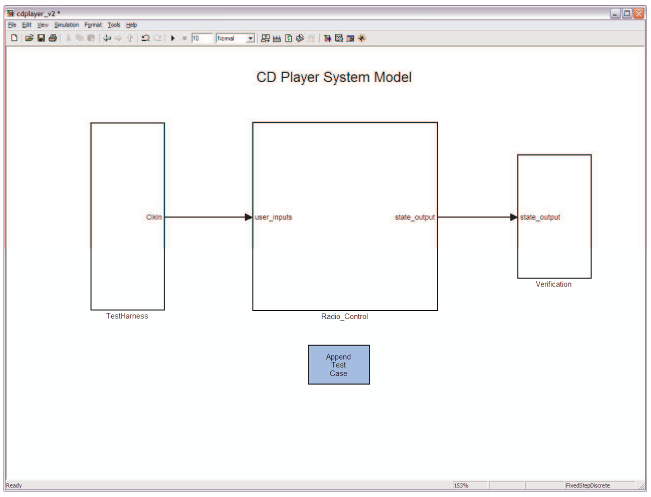

Figure 4: CD player system model.

The MATLAB Graphical User Interface Development Environment (GUIDE) provides a set of tools for creating HMIs. These tools provide an environment to layout and program HMIs.

GUIDE provides users with a palette to drag and drop various input and display elements, such as buttons, text boxes, and gauges. GUIDE ships with a set of commonly used input and display components, but for advanced applications, GUIDE can incorporate images into the HMI and can be extended through the use of the Microsoft ActiveX support when running MATLAB in the Microsoft Windows environment. The ActiveX extension allows users to create or purchase custom input or display components. All the components used to develop the radio face plate in this demonstration are shipped with GUIDE, excluding the graphics.

For the radio face plate HMI, a bitmap image of the physical face plate is used to provide the look and feel of the physical face plate, as well as to create a template as to where buttons and displays need to be placed. Figure 3 displays the bitmap that represents the physical face plate.

For each button that needs to be a testable input to the system, push buttons from the GUIDE palette are placed over top of the button represented in the image. In this case, the ON/OFF button is a desired input and a push button is sized and positioned appropriately to exactly cover the button represented in the physical face plate image, as shown in Figure 5.

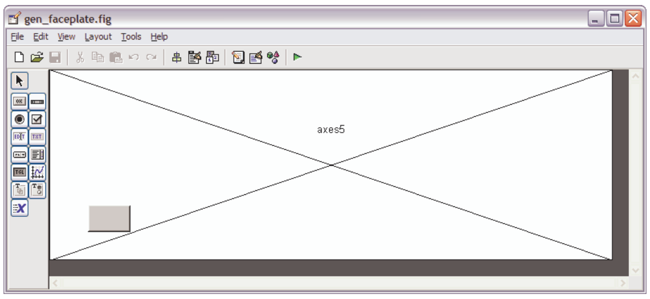

Figure 5: GUIDE push button.

The size and position of the buttons and displays are dynamic and can be altered by a manual drag and drop or via a MATLAB script for programmatic sizing and positioning. When the GUIDE HMI is published, the pushbutton will sit on top of the background image as shown in
Figure 6.

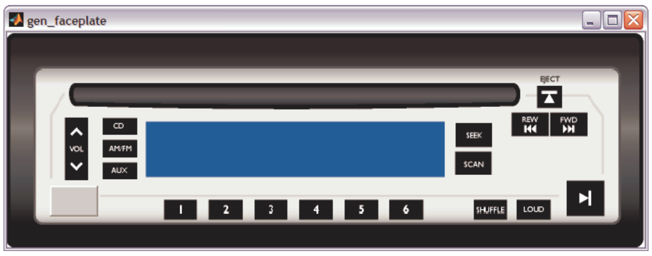

Figure 6: Pushbutton on physical face plate.

To keep the original look of the physical HMI, the button can be skinned with an image as well. For each button that is desired as an input to the system, the section of the image that represents the button is cut and saved in an image file for later use as a skin for the pushbutton in GUIDE. Figure 7 displays the published GUIDE HMI with the skinned ON/OFF image.

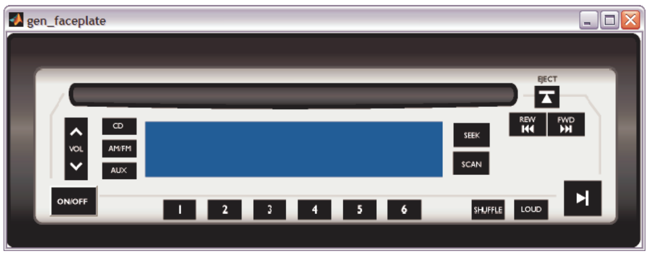

Figure 7: Face plate with skinned ON/OFF button.

The ON/OFF button is now an active button and when pressed by the user, an animation representing the button being pushed down can be noticed. The same procedure can be reused for each button that is desired to represent an input to the system.

Here is the complete set of inputs chosen to be active buttons in this example:

- ON/OFF
- CD
- Eject
- REW
- FWD
- CD Slot
- Battery Connect
- Record button and LED (Used to trigger capturing the button presses)

Figure 8 displays the completed GUIDE layout for the radio face plate that includes all of the buttons positioned and sized based on the set of desired system inputs. Figure 9 displays the published radio face plate HMI and the finished product that the user will interact with to create test cases to exercise the model.

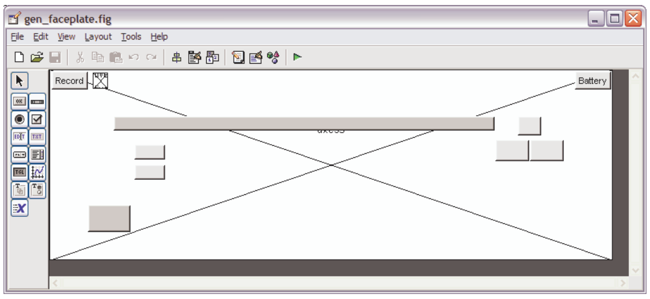

Figure 8: GUIDE face plate example.

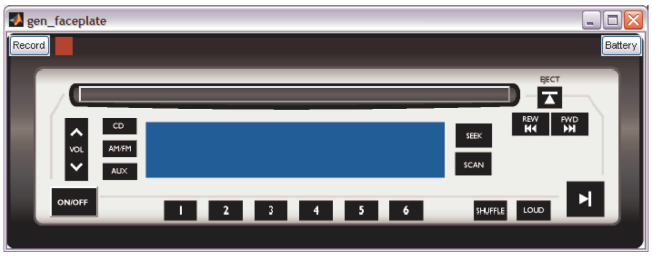

Figure 9: Published radio face plate HMI.

When the HMI is published, GUIDE automatically creates a MATLAB script template that contains callback functions for each active button. Each time a button is pressed, the callback function is called processing the content of the function. The content of the callback function is specified through MATLAB programming providing the capability to perform actions as simple as toggling a value or an even more sophisticated action such as activating a mechanism to capture the user inputs.

As previously mentioned, the overall goal is to be able to create test vectors through interaction with the representative HMI. Essentially, as the user interacts with the HMI, this interaction needs to be captured so the test vectors can be applied to the model for continuous reuse as the model evolves. The MATLAB code shown below demonstrates how to create a time series vector for each button.

```
% --- Executes on button press in
power_pushbutton.
function
power_pushbutton_Callback(hObject, …
  eventdata, handles)


    handles.states.power_request(handles.
    index.power) =…
    ~handles.states.power_previous;
   handles.states.power_previous =…
    handles.states.power_request(handles.
    index.power);

handles.timers.power(handles.index.power)
= toc;
   handles.index.power =
handles.index.power + 1;
```

In this example, we have chosen seven buttons from the radio face plate that are desired inputs to the system so the end result will be seven time series vectors that can be saved for later reuse.

On the upper right hand corner of the HMI, a “Record” button is present with an LED indicator. When the user decides that they would like to create a test vector, the “Record” button is pressed lighting the LED to start the recording process. Pressing the record button merely starts a timer at t=0 and the then system waits for inputs from the user.

Focusing on the ON/OFF button case, the MATLAB code previously shown represents the callback function to capture the change of the ON/OFF button value and time stamp when the change occurred creating the time series vector. MATLAB contains a capability to generate time stamps through the use of the tic and toc commands. Pressing the record button calls the tic command and each button press calls the toc to obtain a time stamp. Since each button on the HMI is a toggle button, the same code snippet can be reused with minimal changes for each active button.

Once the desired sequence of button presses has been completed, simply pressing the record button again will stop the recording process and export each of the seven time series vectors to the Simulink Signal Builder block. The Simulink Signal Builder block contains an API that can be accessed through MATLAB to completely automate exporting the time series vectors to the Signal Builder block. Each time a test case is captured using this code, the newest time series is automatically appended to the existing time series in the Signal Builder block. This automatic appending, allows a suite of test vectors to be created.

Figure 10 shows the Simulink Signal Builder block populated with the user inputs. Also note that there are place holders for the test results, or acceptance criteria, once the model has been exercised with the input signals. Creating the acceptance criteria for each of the tests will be discussed later.

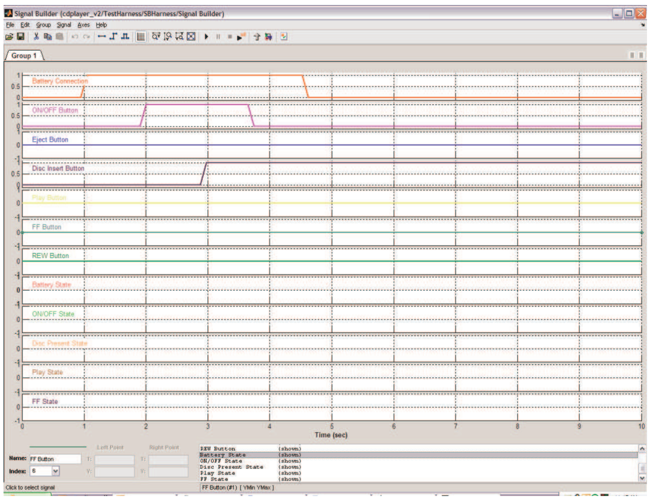

Figure 10: Recorded input test vectors.

Once the Signal Builder block has been populated with the input signals, the system model can be simulated while exercising the system with the inputs that were captured from the HMI. The suite of test vectors can be continuously reused to evaluate the system as the engineer uses the model to iterate through design alternatives. If a design change results in an alteration to a test vector, the engineer could either graphically edit the signals in the Signal Builder block through the Signal Builder interface, or could use the HMI and generate another test case and add it to the test suite, as described in the previous section.

With the ability to create test vectors and simulate the model with these test vectors, the engineer can begin to verify that the system is responding as desired. This desired response is referred to as the acceptance criteria. Unless a design requirement changes, the paired test vector and acceptance criteria should remain constant throughout the design process. At this point, this test case and acceptance criteria will be reused throughout the design process to ensure that no downstream design changes result in failure to meet the requirement.

Essentially those previously mentioned place holders in the Signal Builder block are populated with acceptance criteria – the signals that capture the desired response of the system for a specific set of input test vectors. In this example, the system response is captured by logging when certain states in the CD player state machine become active. When a specific state becomes active, a Boolean TRUE is stored and when the state is inactive, a Boolean FALSE is stored creating a pulse signal. Each state that is monitored is represented by a pulse signal and the entire set of pulse signals characterizes the response of the system to a given input.

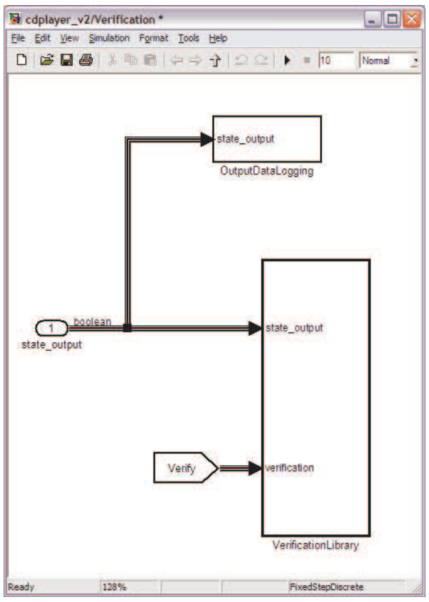

Figure 11: System output logging.

Figure 11 displays the output signal data logging block to capture the response of the system. The Simulink "To Workspace" block is used to store the logged states as vectors in the MATLAB base workspace.

Since the output signals are logged to the MATLAB workspace, similar to exporting the input vectors, using the MATLAB programming language and the Simulink Signal Builder API, the output signals can be post processed and exported to the Signal Builder forming one complete test case. Figure 12 displays the set of input and output vectors forming the complete test case stored in the Simulink Signal Builder block.

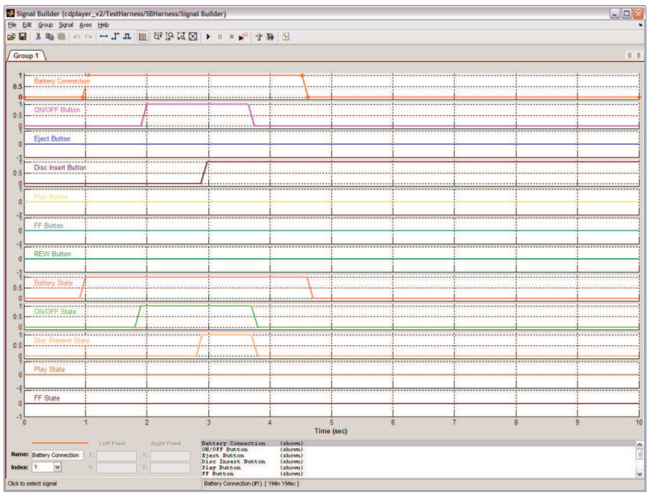

Figure 12: Recorded input and output test vectors.

The same test case can be run on the system model using the previously captured system outputs as a reference for comparison or acceptance criteria in the later tests. In this example, the sequence and duration that the states are active should not change so the "Verification Library", shown in Figure 11, performs a comparison of the current system response to the response stored in the test case.

Other methods for system verification can be performed since elaborate verification methods can be created using other Simulink blocks such as the additional model verification blocks or the base Simulink blocks. For this example, a simple logical comparison of the signals is performed at each time step. The test was created using logical comparisons and Simulink assertion blocks as diagrammed in the "Verification Library" shown in

Figure 13. If the logical comparison blocks produce a false output at any time during the simulation, the Simulink assertion blocks have been set up to detect this false signal, halt the simulation, and display the time step the test failed.

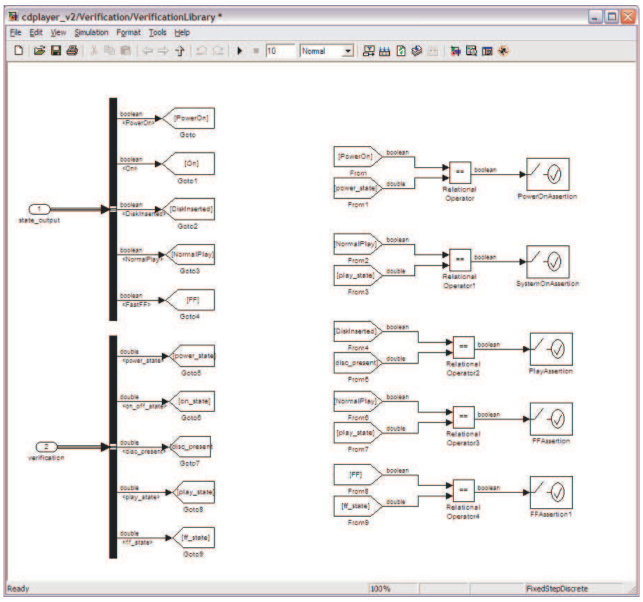

Figure 13: Verification subsystem.

There are a number of assertion blocks that are located in the diagram. The need to check for assertions on only a subset of the output signals is highly probable. A verification manager, which will be discussed later on, allows the user to specify what verification blocks are active for a given test case.

A thorough set of test vectors provides the ability to identify changes in the system response and quickly flag when a change in the model results with a requirement that is no longer being met. This discussion leads to the topic of requirements-based testing.

We described earlier that paper requirements specifications typically specify how a system should respond based on a set of inputs to the HMI. These paper requirements form the basis for how the entire system is designed and tested. Developing test cases from the requirements and traceability to the requirements document is a powerful capability as the design matures.

Given that a representative HMI can be created, it is natural to go through the requirements specification and start developing test cases at the model level following the procedure detailed in this paper. A common CD player requirement is the following:

Depressing the CD button when a readable disc is present in the CD mechanism, the system shall enter the 'Play' State.

Based on this requirement, a test case can be created using the HMI to create the input vectors, and the response of the system can be analyzed. If the system responds as desired, the system response will be added to the test case and saved for later reuse in the Signal Builder block. Figure 14 displays the test case created to test the CD button requirement.

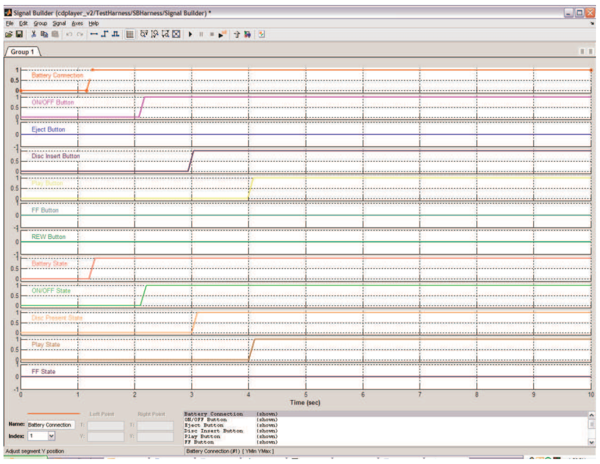

Figure 14: Test case based on a requirement.

In HMI-dependent systems, the number of tests required to fully cover the model can be quite large. Requirements traceability provides a method to specify which requirements are being tested for a given test case; if a test does fail, the requirements that are no longer met are immediately evident.

Using Simulink Verification and Validation [8], the documented requirements can be linked within the Signal Builder block to a specific test case or a group of test cases. Selecting the Simulink Verification and Validation icon in the Signal Builder interface opens the verification and requirements managers. The verification manager allows the user to specify what verification blocks are active for a given test case. The requirements manager allows the user to specify what requirements are linked to the given test case.

Figure 15 displays the Signal Builder interface with the verification and requirements managers displayed. If you look closely to the requirements on the right hand side, the link to the CD button requirement is present. When that link is selected, the requirements document is automatically opened and the linked requirement is displayed.

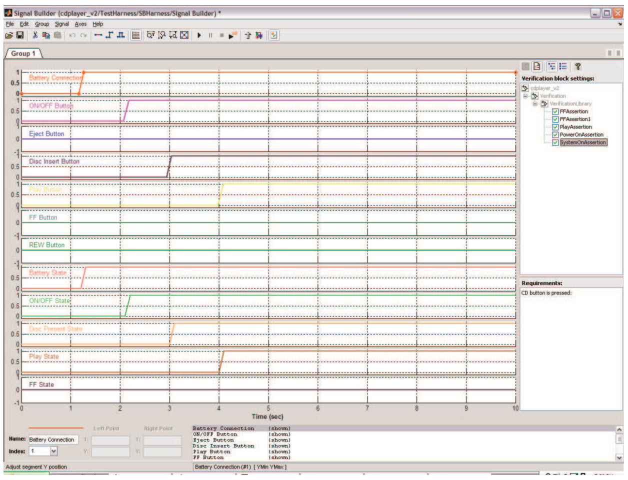

Figure 15: Signal Builder with the verification and requirements managers.

The Signal Builder block incorporates the capabilities to manage the test signals, verifications, and requirements under one interface. Using the verification and requirements managers allows an engineer to create effective test cases for easy reuse as the model is elaborated. The HMI provides an efficient means to populate the signals in the signal builder block, and the requirements can be easily added to complete the test case providing traceability back to the requirements document.

CONCLUSION

The prevalence of electronic devices with complex Human Machine Interfaces is growing every year. The need to provide early verification and validation of the algorithms and the interfaces is a challenge for engineers who traditionally wait for hardware to begin testing. In this paper, we have presented a process by which engineers can create a graphical representation of the interface, recorder their tests, and use them with models of the algorithm to verify the design.

REFERENCES

1. www.mathworks.com/applications/controldesign/description
2. The MathWorks Inc., "*Using MATLAB*," Version 7.3, The MathWorks Inc., Natick, MA, September, 2006.
3. The MathWorks Inc., "*Using Simulink*," Version 6.5, The MathWorks Inc., Natick, MA, September, 2006.
4. Thomas Sedran, Thomas Wendt, and Antonio Benecchi, "*Electronics & Automotive: Achieving a more solid Union,*" Automotive Design & Production, May 2005.
5. The MathWorks Inc., "*Stateflow User's Guide*," Version 6.5, The MathWorks Inc., Natick, MA, September, 2006.
6. The MathWorks Inc., "*Creating Graphical User Interfaces*," Version 7.3, The MathWorks Inc., Natick, MA, September, 2006.
7. The MathWorks Inc., "*Using Simulink*," Version 6.5
8. The MathWorks, Inc., "*Simulink Verification and Validation User's Guide*," Version 2, The MathWorks, Inc., Natick, MA, September 2006.

CONTACT

Chris Fillyaw
Sr. Applications Engineer
(248)596-7925**,** Chris.Fillyaw@mathworks.com**.**

Chris has been involved in developing automotive systems for over seven years and has been leveraging the capabilities of The MathWorks tools throughout his career. Chris is based out of the Detroit, Michigan office where he focuses on working with automotive customers who are interested in adopting Model-Based Design. Chris graduated from Michigan Technological University with Bachelors in Electrical Engineering and received his Masters in Electrical Engineering from The University of Michigan – Dearborn.

Jonathan Friedman
Automotive Industry Marketing Manager
(508) 647-7752, Jon.Friedman@mathworks.com.

Jon leads the marketing effort to foster industry adoption of The MathWorks tools and Model-Based Design. Before joining The MathWorks, Jon held various positions at Ford Motor Company that included working on software development research at the Ford Research Lab, working in Product Development as a Vehicle Launch Leader at plants across North America, and as an Electrical Engineering Supervisor. Jon has also worked as an Independent Consultant on projects for Delphi, General Motors, Chrysler and the US Tank-automotive and Armaments Command. Jon holds a B.S.E., M.S.E. and Ph.D. in Aerospace Engineering as well as a Masters in Business Administration, all from the University of Michigan.

Sameer M. Prabhu
Sr. Applications Engineering Team Lead
(248) 596-7944, Sameer.Prabhu@mathworks.com.

Sameer has over ten years of experience applying The MathWorks products in various application areas. As a Senior Team Lead in the Detroit, MI office, Sameer manages a team of applications engineers focused on working with customers in the automotive and commercial vehicle industry to address the systems integration challenges posed by increased adoption of electronics in these industries. Sameer graduated from University of Bombay with Bachelors in Mechanical Engineering and received his Ph.D. in Mechanical Engineering from Duke University in the area of robotic controls and artificial intelligence. He also holds an MBA from The University of Michigan.