2008-01-1052

Testing Human Machine Interface (HMI) Rich Designs using Model-Based Design

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

ABSTRACT

Today's vehicles are typically outfitted with passenger convenience features that require Human Machine Interfaces (HMIs). HMIs can be relatively simple – such as a remote key fob – or more sophisticated – such as a radio face plate. Traditional development of HMIs involves two typically independent processes – (1) Physical Component Design and (2) Functional Logic Design. The physical component design is developed by a team that usually includes both graphics and ergonomics designers to ensure that the HMI is intuitive and fits well with the interior styling of the vehicle. The functional logic design follows a more typical software development process. This process is based on functional requirements commonly written in terms of user requests and system responses as represented by the HMI. As the complexity of the system increases, it is essential for the intuitiveness and ease of use of the HMI to advance as well. For teams using traditional methods to design, prototype, and fully test an HMI, achieving this level of ease of use is becoming increasingly challenging. In an earlier paper, the authors demonstrated how to create a "soft" version of the HMI under development and then use this to generate and record test vectors [1]. These test vectors could then be used to exercise the design under test to determine if the HMI logic was completely tested and if the design met the specified requirements. In this paper, the authors will address two important workflow issues to support the logic design and verification of HMIs – (1) integration of the HMI graphics and HMI logic and (2) formal verification of the HMI logic.

INTRODUCTION

It is estimated that electronics and software content will make up 40% of a vehicle's cost by the year 2010 [5]. In the past, the electronic systems that replaced conventional mechanical systems were used primarily to ensure that vehicles met stringent emissions and fuel economy requirements. Today, automotive manufacturers are expanding the use of electronics to introduce advanced multimedia and convenience features to attract technology-savvy buyers. These features assist the driver by providing relevant information such as up-to-date traffic information to change the planned route to a destination. Technologies such as hands-free phone via the car's radio and wireless networking make these convenience features possible. In addition to assisting the driver, these systems can 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 way to differentiate themselves from the competition, and also as the basis of a lucrative revenue stream. As a result, there is an increased emphasis on developing and deploying these systems to meet consumers' varying requirements. At the same time, the systems must be simple enough to operate and use easily and they must meet high quality standards to avoid costly recalls and software fixes.

Figure 1 - Virtual Radio Faceplate HMI

MODEL-BASED DESIGN FOR MULTIMEDIA AND CONVENIENCE FEATURES

In an earlier paper the authors discuss the use of Model-Based Design to address the challenges of increasing product complexity, more stringent performance requirements, and shorter product development cycles [1]. The use of Model-Based Design to address these challenges was demonstrated using the radio faceplate as an example. The paper described the creation of a "soft" version of the HMI under test to generate and record test vectors. These test vectors could then be used to exercise the design under test to determine if

the HMI logic was completely tested and if the design met the specified requirements. The process outlined in the paper was as follows:

1. Develop the “soft” HMI representation.
2. Capture the user inputs to the HMI and populate a set of test vectors.
3. Exercise the model with the test vectors, capture the system response and analyze whether the response meets the design requirements.
4. Edit test vectors as needed based on requirements specifications.

The previous paper built a foundation for improving the process by which HMI rich systems are designed and verified. With a realistic virtual HMI in place, the engineer can connect it to a model of the logic and then exercise the logic as the end user would. The physical HMI design places many constraints on the functional logic design. The ability to test the logic while developing it can quickly highlight any deficiencies in the requirements and design assumptions.

With a set of test vectors created during the design process based on the requirements and design assumptions, formal verification of the logic through automatic test vector generation can reveal previously unconsidered test scenarios. Formal verification can also ensure that a complete test suite fully covering the model can be developed, enabling the engineer to exhaustively test the HMI rich system well before hardware prototypes have been constructed. This paper discusses the importance of using the virtual HMI to test the system during development, connecting the virtual HMI to the model, and using formal verification and validation techniques to exhaustively test the system.

The paper includes:

a detailed description of the benefits of exercising an HMI rich design with a realistic virtual HMI and a step-by-step process for connecting the virtual HMI to a Simulink® model

a discussion of formal verification and validation techniques using automatic test vector generation for analyzing model coverage

INTEGRATION OF HMI GRAPHICS AND FUNCTIONAL HMI LOGIC

The HMI of the system is the sole interface through which the user perceives and interacts with the state of the system. As detailed in the earlier paper, a key aspect of the multimedia and convenience features is ease-of-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.

The intuitiveness and ease-of-use of a system can only be determined by users actually interacting with the physical system or a representative virtual system. The ability to interact with this logic as the customer would allow system testing based on functional requirements. When these requirements, which are often written in terms of the customer interaction with the interface device, can be tested as engineers are developing the HMI logic, the team can asses the intuitiveness and ease-of-use of the HMI while evaluating the requirements early on in the development process.

Integrating the graphics with the logic supports design exploration of the logic as well as both technical and managerial reviews of logic design. The virtual HMI developed in the previous paper as a means to create test vectors can also be linked to the logic model using a scripting language. To link the soft HMI to the logic, each HMI element needs element-specific code in the overall logic and graphic code.

Because the development of the virtual HMI was covered in the previous paper, the details of constructing the soft HMI will not be covered again here. This section describes the connection of a virtual HMI of the radio faceplate to the functional logic that was created in Simulink. This connection enables the model to be exercised and accessed during the simulation of the functional logic and allows the user to perceive a realistic representation of the physical system.

Figure 1 shows the virtual radio faceplate HMI that was described in the earlier paper and created with GUIDE, the MATLAB® graphical user interface (GUI) development environment. This virtual HMI is a realistic representation of the physical radio faceplate with all of the buttons and indicators that customers would have access to in their vehicle. The virtual HMI from the earlier paper was modified to include an active LCD display and to remove the record button, which is no longer required when the HMI is connected to the Simulink model.

The virtual HMI provides an interface to the functional logic which has been modeled using both Simulink and Stateflow®. Stateflow enables engineers to model the state driven machines and discrete event reactions needed in HMI rich designs. While a detailed discussion of the underlying logic is beyond the scope of this paper, a review of the top level model architecture will help to illustrate how the HMI fits into the system.

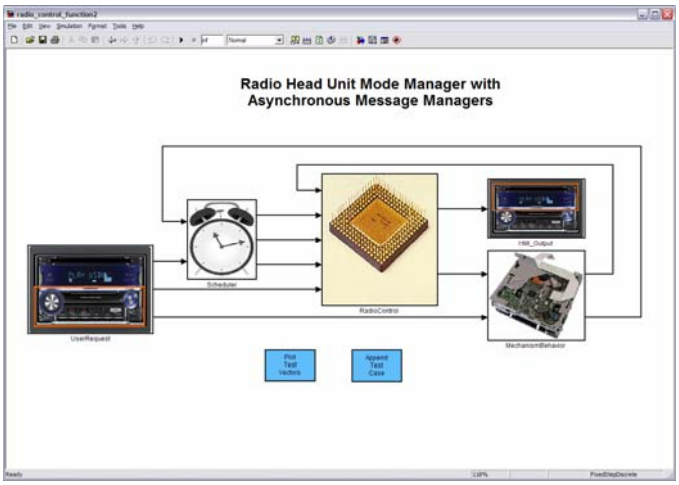


Figure 2 - Radio faceplate model.

Figure 2 shows the top level Simulink model and the components that make up the system. The model contains two main components, a mode manager and a message manager, both called by a scheduler. The mode manager is the functional logic component and it is executed at the base rate of the model. The message manager is only executed when a user presses a button on the virtual HMI. The model also includes input and output interfaces used to receive button presses from the virtual HMI and update the LCD display.

The button presses on the virtual HMI are captured and translated into a message that is transmitted to the message manager. In this design, which is similar to how an actual radio might be architected, the message manager receives the messages from the virtual HMI input interface and translates them into commands that are then sent to the mode manager. The mode manager does not have a separate input for each button press since it receives a message containing a payload with the requested mode change.

Because the mode manager is modeled as it would be implemented on real hardware that interacts using communication buses, a button press is received by the mode manager as a request and an 8-bit value instead of a unique Boolean input. This messaging level of abstraction makes it difficult to test the system without a virtual HMI.

To link the soft HMI and the logic developed, each HMI element will need additional element-specific code. To better understand the process for connecting a button to the model, consider the ON/OFF button as an example.

When the ignition key is in the ON or Accessory position, the radio head unit is typically powered and is waiting for a request from the user – for example, turning the radio on or selecting a desired mode (such as AM, FM, or CD). The functional logic behind the HMI accepts inputs from the HMI, performs the decision-making logic to handle the user request, and displays any change in radio state. An example of the functional logic for the ON/OFF button modeled using Stateflow is shown in Figure 3.

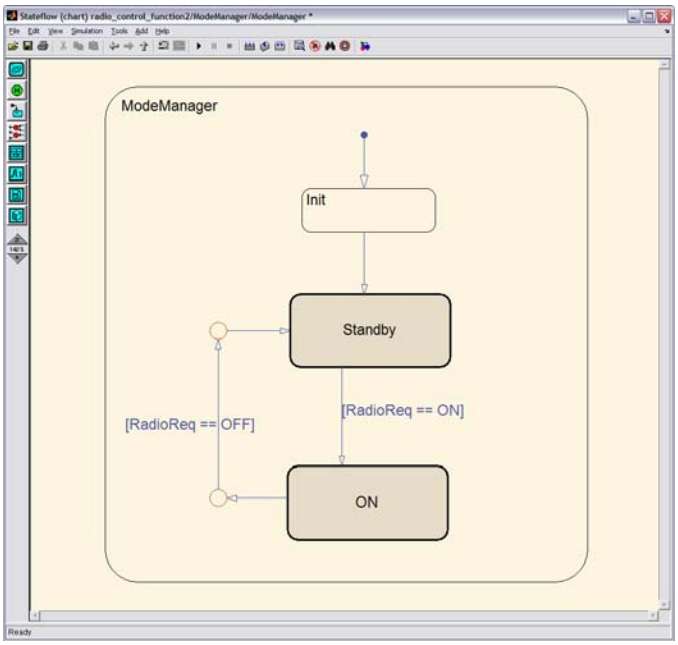


Figure 3 - HMI functional logic modeled in Stateflow®.

When the radio is first powered up, it enters the standby state by default and waits for a user to press the faceplate ON/OFF button before entering the ON state with the appropriate audio mode. Using the virtual HMI, the engineer can exercise the model just as the requirement is written to ensure the design fulfills the requirement.

The engineer can also watch the indicators on the virtual HMI to determine if the radio has entered the ON state and is in the desired mode. Figure 1 shows the virtual HMI when the radio is in the Standby state and Figure 4 displays the virtual HMI when the radio head unit is in the ON state with FM mode selected (as indicated by the LCD display).

Figure 4 - Virtual HMI FM mode state.

This is an example of exercising the model through the virtual HMI to test the logic and ensure that it is functioning as expected and meeting the requirements. In the event that the system does not respond properly to the button presses as described by the requirements, model animation capabilities enables the engineer to

investigate visually the reaction of the internal HMI logic to the button presses. In Figure 5, the FM state is highlighted in blue designating this as the active state during simulation. Highlighting provides the engineer with immediate feedback on the system's response to button presses.

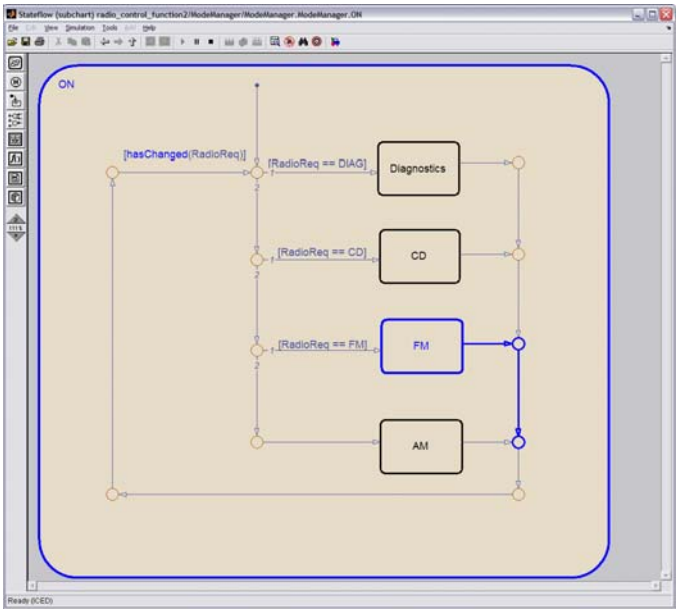


Figure 5 - Stateflow® animation.

The virtual HMI enables engineers to evaluate ease-of-use by analyzing end user interactions with the system. Likewise, the ability to interact live with the system model during simulation provides valuable insight during development and throughout system verification and validation.

In the previous paper, MATLAB GUI-building tools were used to layout and generate the virtual HMI with a number of pushbuttons allowing the engineer to stimulate the model. For this paper, an LCD display was added to display the state of the system because the buttons now affect the functional logic model. Gauges Blockset™ provides display components, including the LCD display used in this example that can be accessed from MATLAB GUIs.

The virtual HMI can be connected to the model using the following process:

1. Identify the inputs and outputs of the system that the virtual HMI will control and display.
2. Add a unique source block to the model to correspond to each input controlled by the virtual HMI.
3. Translate the virtual HMI button presses to the source block values using MATLAB code.
4. For each display element of the HMI, create unique sink blocks that accept model outputs to be displayed using MATLAB code.

Each of these steps will be explained in detail within the context of a radio head unit model.

IDENTIFYING INPUTS AND OUTPUTS

The radio faceplate virtual HMI contains 21 different buttons that can be used as system inputs. This example focuses on the ON/OFF button and describes the steps needed to connect it to the Simulink model. The same process can be applied to each of the 21 buttons.

Because the sole indicator on the radio faceplate is the LCD display, only one output from the system is required. In this paper, only the ON/OFF button will be connected to the Simulink model. With the input and output identified, step one of the process is complete.

ADD SOURCE BLOCKS

In Simulink all signals or inputs to a system must be driven by a source. One method for connecting the virtual HMI to the model is to add a Constant block for each input signal to the model. By modifying the automatically generated function callbacks in the MATLAB GUI code, an engineer can change the value of the Constant block programmatically when a button is pressed. Figure 6 shows the Constant blocks added to the front end of the Simulink model to drive the system inputs. To allow the HMI to alter the value of a Constant block, the Inline parameters on the Optimizations pane of the Constant block's Configuration Parameters dialog *must be unselected*.

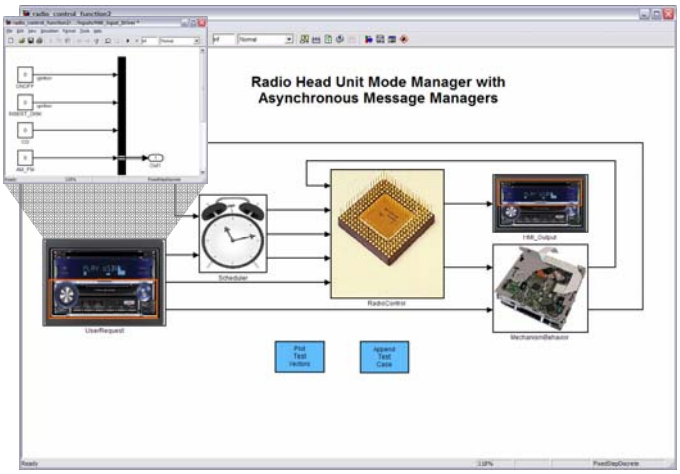


Figure 6 - Virtual HMI Constant blocks.

ADDING MATLAB CODE

With the Constant blocks in place, the MATLAB code to change the value of the Constant block can be added to the existing function callbacks automatically generated by GUIDE. To link the ON/OFF button from the HMI to the functional logic, add the following code to the existing ON/OFF function callback:

```
% --- Executes on button press in power_pushbutton.
```

```
function power_pushbutton_Callback(hObject, eventdata, handles)
% hObject    handle to power_pushbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
   power_current_state =
get_param('radio_control_function2/UserRequest/Inputs/HMI_Input_Driver/
ONOFF', 'Value');

set_param('radio_control_function2/UserRequest/Inputs/HMI_Input_Driver/O
NOFF', 'Value', num2str(~str2num(power_current_state)));
```

In this example, the code retrieves the current state of the ON/OFF Constant block using the get_param() function, and toggles the value using the set_param() function.

The same technique can be used for each of the buttons in the virtual HMI.

CONTROLLING THE VIRTUAL HMI LCD

With the inputs mapped to the Constant blocks in the Simulink model, the model can be excited through the use of the virtual HMI. During simulation, pressing the buttons on the virtual HMI will cause the states to execute the appropriate transition for each button press. Technically, this is enough functionality to exercise the model and perform interactive testing of the model while visualizing state transitions in Stateflow charts. To fully verify that the functional logic is transmitting the appropriate output messages or toggling the appropriate indicators, the output of the model should be connected to the LCD display in the virtual HMI.

There are a number of methods to control the virtual HMI LCD and display the various messages specified by the functional requirements. The user interface displays can all be accessed from the MATLAB workspace. Using a handle to the virtual HMI that is provided when the interface is instantiated, an engineer has complete control over each of the display components of the user interface. The model can be set up to automatically open the appropriate virtual HMI when the model is opened and attach a handle to the user interface for controlling the LCD display component.

When the user interface is instantiated, a handle to the user interface is provided as a return value. In the radio faceplate example, the virtual HMI was created in GUIDE and named “gen_faceplate”. GUIDE automatically creates a MATLAB script with all the callbacks. When executed, this script will instantiate the user interface and return a handle to the user interface. The following two MATLAB commands can be used to instantiate the user interface and obtain its handle:

```
% Instantiate GUI and create a handle

h = gen_faceplate;          % instantiate the user interface
hGUI = guihandles(h);    % create a structure of user interface handles
```

Simulink can use these handles through MATLAB and, as in this example, assign various sets of string data to the LCD display. One way to do this is using a MATLAB Function block in Simulink that calls a MATLAB function from within Simulink for each time step during the simulation.

The Simulink model uses a MATLAB set() function in the MATLAB function block to control the text displayed on the LCD display in the virtual HMI.

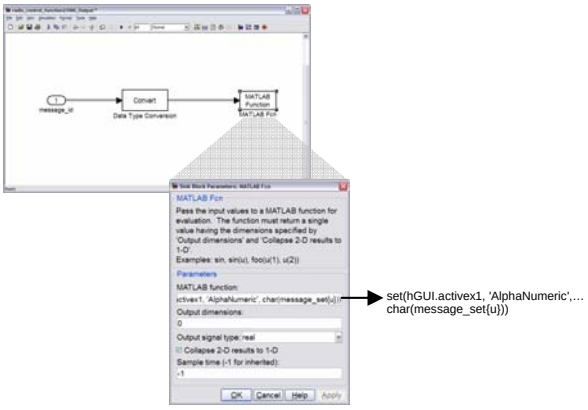


Figure 7 - MATLAB® function block controlling LCD display.

Similar to how a message definition set may be implemented in C code, a MATLAB cell array is created that stores a set of messages to be displayed on the LCD. Stateflow logic keeps track of which message to display as an index value into this cell array. This cell array is referenced in the MATLAB Function block, as shown in Figure 7. The MATLAB code that defines the message set is shown below.

```
% define message set
clear message_set;

message_set{1} = '';
message_set{2} = 'ON';
message_set{3} = 'NO DISC';
message_set{4} = 'DISC';
message_set{5} = 'PLAY';
message_set{6} = 'REWIND';
message_set{7} = 'FF';
message_set{8} = '101.1 FM';
message_set{9} = '1050 AM';
```

With the realistic virtual HMI implemented, the engineer can interact with the system enabling rapid test case generation to ensure the design meets the requirements and the design intent. Similar to the test case logging described in the previous paper, the engineer can use the virtual HMI to interact with the model, storing the sequence of button presses as signals using the Signal Builder source block in Simulink for testing in the future.

The process in this case is somewhat simpler, because the connection between the virtual VMI and the Simulink model streamlines signal logging. Simulink provides an easy method to log signals to the MATLAB workspace. Once the signals are logged to the workspace, the Signal Builder API, as described in the earlier paper, can

be used to populate Signal Builder with the test case to be saved. Signal Builder blocks can then be used to store the entire suite of test cases for running the complete test suite against the model without interaction from the user. Engineers can log the outputs and later visualize and the compare the results.

FORMAL VERIFICATION OF THE HMI FUNCTIONAL LOGIC

Once the HMI logic design is complete, test engineers can create a set of reusable test vectors using the process discussed in the previous paper and highlighted above. However, given the combinatorial complexity of the current set of automotive HMIs, it is nearly impossible for test engineers to fully exercise all the logic in the design.

For example, given a set of m buttons on a typical HMI in which each button can lead to n different sub-menus with a different set of functions for each button, and where "no function" is also considered a function to be verified, a test engineer would have to write m^n tests. If a combination of buttons enables other functionality, such as diagnostics, then the set of tests becomes even more complicated. Clearly, guaranteeing that the logic is fully tested goes well beyond the functional testing specified by the requirements and what can be feasibly accomplished by manually pressing all of the button combinations.

As an example, the requirement that "when the battery is powered (key in Accessory or ON position), the audio unit goes into the PowerOn mode," also requires that all of the buttons be active. Moreover, when the radio is in PowerOff mode, the unwritten requirement is that all of the buttons need to be inactive.

Using tools that take advantage of formal methods such as Simulink® Design Verifier™ [9], engineers can analyze the logic models to determine the inputs that will result in full functional testing of the logic and identify unreachable states (such as switch conditions that cannot occur) [9].

Using software that implements formal methods, engineers can generate tests for models to satisfy model coverage and user-defined objectives. Engineers can also prove model properties and generate examples of violations.

For this paper, we are using Simulink Design Verifier, which supports the following model coverage objectives: decision, condition, and modified condition/decision coverage (MC/DC). Additionally, one can define custom test objectives directly using modeling constructs available in the modeling packages, such as the design verification blocks in Simulink and Stateflow. With property proving, engineers can explore their design for flaws, missed requirements, and unwanted states, issues that are difficult to uncover by simulation.

The radio faceplate example has seven buttons that have been made active in the virtual HMI, which makes it a significant challenge to manually create the required number of test cases to completely test the specification. In addition, there are a number of test case possibilities that are not covered by the requirements or may not have been considered by the designer.

Requirements-based testing is a necessity and is an effective first pass when developing test vectors, but there are many button press sequences or internal events that are not explicitly covered in the requirements and that may cause undesirable behavior of the radio. These conditions are the most difficult to test and can go undetected until the radio has been placed in the vehicle for testing on a bumpy road where it is common for the driver to press buttons out of order. For example, when inserting a CD, the user may accidentally press the fast-forward (FF) button before the CD has been fully inserted. A typical test engineer developing tests to the requirements specification would insert the disc and wait for it to start playing before pressing the FF button. In the real world scenario, because the FF button was pressed prior to the disc being fully inserted, the behavior of the radio for this unexpected condition is not known.

Formal methods can assist in finding these undesirable, unintended conditions. Many of these scenarios go untested when using manual test vector generation techniques driven by the requirements simply because it is very time consuming and difficult to think of these abnormal button presses. Formal verification methods and automatic test vector generation enable engineers to analyze the system as an independent party that has no preconceived notions of how the system is designed. Formal verification techniques with automatic test vector generation analyze the system and automatically generate test vectors based on the possible decision transitions contained in the system.

In the HMI model, the mode manager contains all of the functional logic. As a result it makes sense to rigorously test this component to achieve close to 100% model coverage and determine if there are any unreachable states within the model. In this example, the mode manager is an atomic subsystem; by right clicking on the subsystem block and execute Simulink Design Verifier, engineers can automatically generate tests as shown in Figure 8.

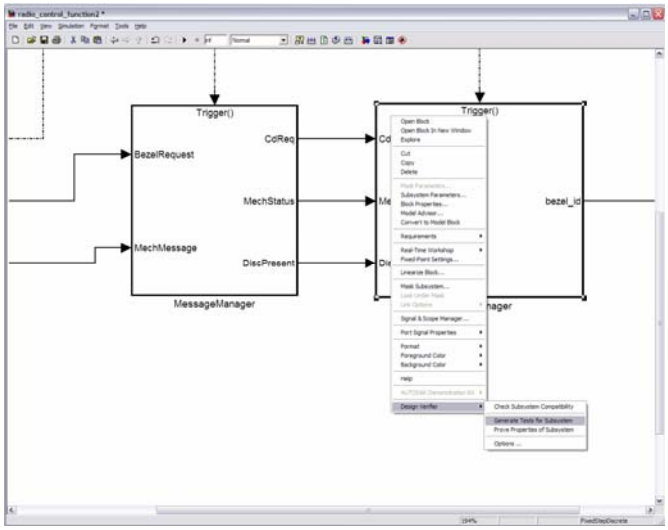
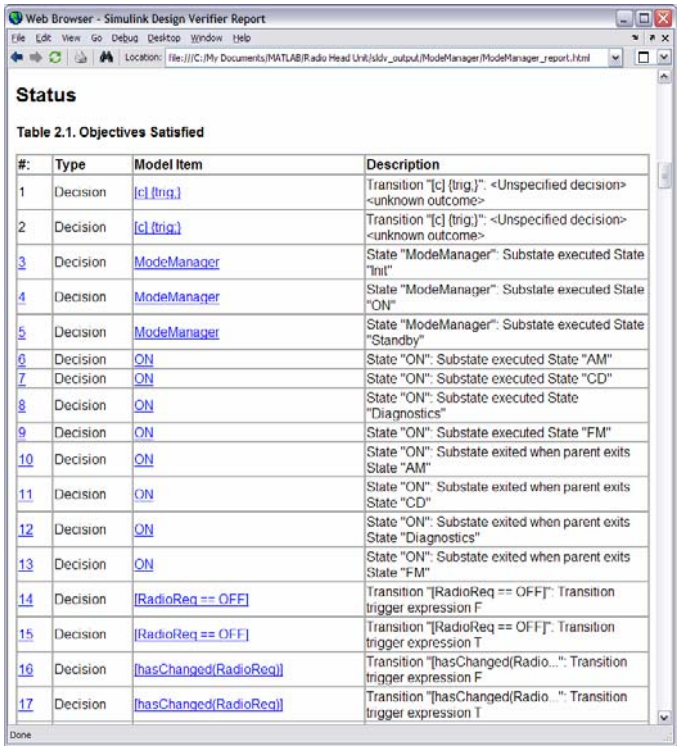


Figure 8 - Executing Simulink® Design Verifier™ on a subsystem.

Once the portion of the system to be tested is selected, the model will be analyzed and a list of test objectives is created to achieve the model coverage objectives selected by the engineer. In the radio faceplate case, decision coverage is all that is required given the nature of the functional logic in the mode manager subsystem as shown in Figure 9.


Web Browser - Simulink Design Verifier Report

Status

Table 2.1. Objectives Satisfied

#:	Type	Model Item	Description
1	Decision	[c] {trig;}	Transition "[c] {trig;}": <Unspecified decision> <unknown outcome>
2	Decision	[c] {trig;}	Transition "[c] {trig;}": <Unspecified decision> <unknown outcome>
3	Decision	ModeManager	State "ModeManager": Substate executed State "Init"
4	Decision	ModeManager	State "ModeManager": Substate executed State "ON"
5	Decision	ModeManager	State "ModeManager": Substate executed State "Standby"
6	Decision	ON	State "ON": Substate executed State "AM"
7	Decision	ON	State "ON": Substate executed State "CD"
8	Decision	ON	State "ON": Substate executed State "Diagnostics"
9	Decision	ON	State "ON": Substate executed State "FM"
10	Decision	ON	State "ON": Substate exited when parent exits State "AM"
11	Decision	ON	State "ON": Substate exited when parent exits State "CD"
12	Decision	ON	State "ON": Substate exited when parent exits State "Diagnostics"
13	Decision	ON	State "ON": Substate exited when parent exits State "FM"
14	Decision	[RadioReq == OFF]	Transition "[RadioReq == OFF]": Transition trigger expression F
15	Decision	[RadioReq == OFF]	Transition "[RadioReq == OFF]": Transition trigger expression T
16	Decision	[hasChanged(RadioReq)]	Transition "[hasChanged(Radio...": Transition trigger expression F
17	Decision	[hasChanged(RadioReq)]	Transition "[hasChanged(Radio...": Transition trigger expression T

Figure 9 - Simulink® Design Verifier™ test objectives.

Each of the identified test objectives corresponds to decision logic in the mode manager and will require a test case that can satisfy the logic for both the TRUE and FALSE case in order to be fully covered. Condition coverage and MCDC are more complicated coverage metrics, but they are not required in this example given the logic in the model. Simulink Design Verifier will attempt to generate a test case to satisfy each of the test objectives if it is possible, meaning there are no unreachable states or unachievable transitions.

Any states or transitions which are unreachable or are not able to be tested given the definition of the standard coverage metrics will be reported to the engineer. It is not necessarily imperative to obtain 100% model coverage, even though it should be the goal, it is more important to be able to understand why there might be cases where 100% model coverage is not achievable. In these cases, the engineer will need to evaluate whether there is a deficiency in the design requiring a change or if the lack of coverage is expected.

For the radio head unit, 21 test cases were generated to test all of the identified objectives. After generating the test cases, a new test harness model is automatically generated. The system under test is copied into this model and connected to a Signal Builder block containing each of the automatically generated test cases. Figure 10 shows the automatically generated test harness and Figure 11 shows the Signal Builder block containing each of the automatically generated test vectors.

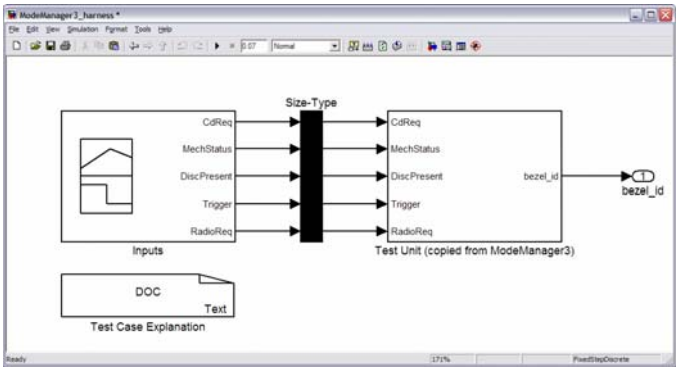


Figure 10 - Automatically generated test harness.

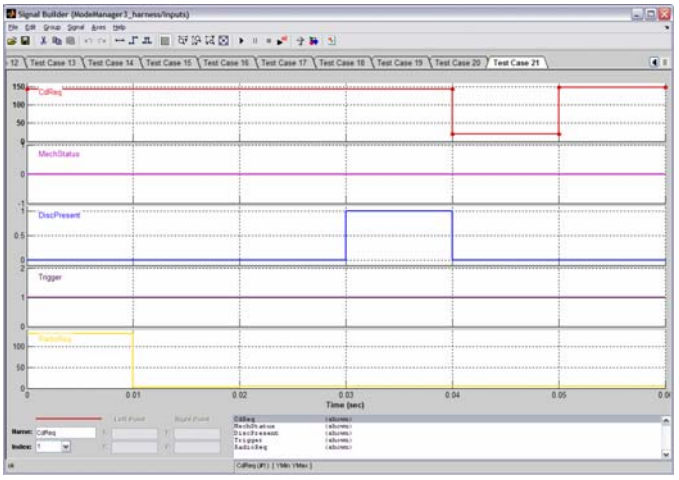


Figure 11 - Automatically generated test vectors.

A common technique is to merge the test vectors that were previously generated using the recorded virtual HMI button presses with the automatically generated test vectors. For this example, the two sets of test vectors can be combined using the Signal Builder API in Simulink. The MATLAB code to merge two signal builder blocks is shown below.

```
function merge_sig_build(SigBuild1, SigBuild2)
%
% Merge the contents of two signal builder blocks
%
% SigBuild1 = destination signal builder block where the contents of
%
% SigBuild2 = additional test vectors to be merged with SignBuild1
%

%% Determine number of groups to be copied from the signal builder
% block
[sbTime,sbData, sbSigLabels, sbGroupLabels] =
signalbuilder(SigBuild1);
NumGroups = size(sbData,2);

signalbuilder(SigBuild2, 'append',sbTime, sbData, sbSigLabels,
sbGroupLabels);
```

With the test suite containing both the requirements based test cases and the automatically generated test cases, the engineers can be confident that they are exhaustively testing their system and clearly understand how it will perform prior to it being implemented on the hardware. With the goal of achieving 100% model coverage with the test suite, requirements based test and formal methods achieved the full 100% decision coverage as displayed in Figure 12.

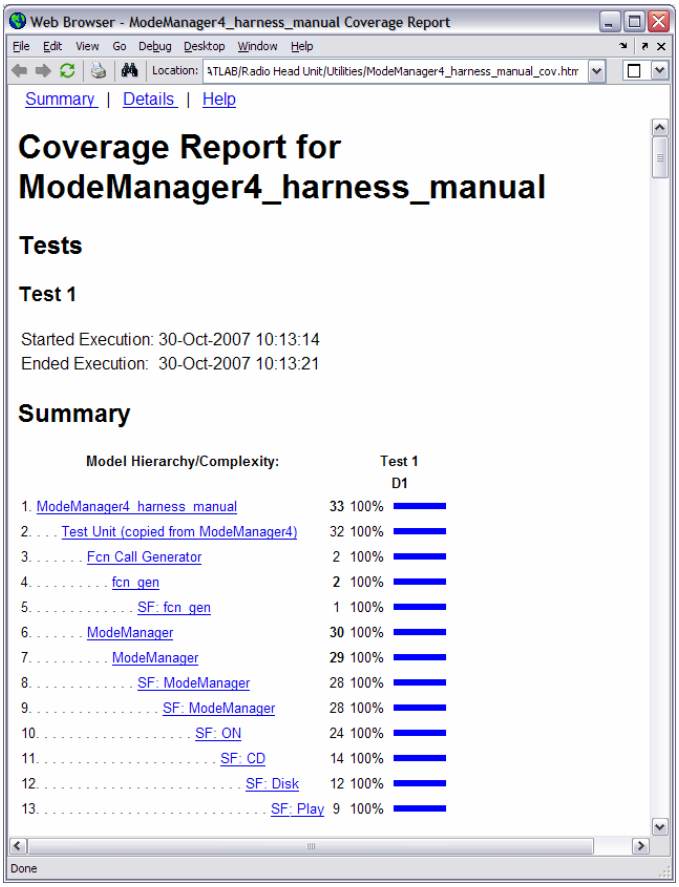


Figure 12 - Radio head unit model coverage report.

CONCLUSION

Using both functional and formal verification techniques, automotive engineers can develop their HMI designs with confidence that random button presses will not result in unintended functionality of the device from untested logic. This paper demonstrates how engineers can connect a virtual HMI to a Simulink model and interactively test and asses the functionality of the HMI logic. Considering the complexity of the functional logic and the time needed to fully test an HMI rich design, formal verification techniques can be applied to identify unreachable states or transitions which cannot be satisfied using the inputs to the system.

REFERENCES

1. Chris Fillyaw, Jonathan Friedman, Sameer M. Prabhu, "*Creating Human Machine Interface (HMI) Based Tests within Model-Based Design*", SAE Paper 2007-01-0780.
2. www.mathworks.com/applications/controldesign/description
3. The MathWorks Inc., "*Using MATLAB*," Version 7.5, The MathWorks Inc., Natick, MA, September, 2007.
4. The MathWorks Inc., "*Using Simulink*," Version 7.0, The MathWorks Inc., Natick, MA, September, 2007.
5. Thomas Sedran, Thomas Wendt, and Antonio Benecchi, "*Electronics & Automotive: Achieving a more solid Union,*" Automotive Design & Production, May 2005.
6. The MathWorks Inc., "*Stateflow User's Guide*," Version 7.0, The MathWorks Inc., Natick, MA, September, 2007.
7. The MathWorks Inc., "*Creating Graphical User Interfaces*," Version 7.5, The MathWorks Inc., Natick, MA, September, 2007.
8. The MathWorks, Inc., "*Simulink Verification and Validation User's Guide*," Version 2.2, The MathWorks, Inc., Natick, MA, September 2007.
9. The MathWorks, Inc., "*Simulink Design Verifier User's Guide*," Version 1.1, The MathWorks, Inc., Natick, MA, September 2007.

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
Applications Engineering Manager
(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.