# Challenges in Automated Model-Based HMI Testing

Reinhard Stolle, Thomas Benedek, Christian Knuechel, Harald Heinecke

BMW Car IT
Petuelring 116
80809 München
reinhard.stolle@bmw-carit.de

**Abstract:** We describe our approach to automated model-based HMI testing. The paper is divided into two parts. In the first part, we summarize the current status of our work. In the second part, we describe a number of research areas that need to be worked on in order to achieve true model-based HMI test automation.

## 1 Test Automation in the HMI Domain

The task of **test automation** involves two subtasks: (1) automated test case selection, and (2) automated test case execution. The long-term goal of test automation is to test against a complete formal specification of the unit under test. To be useful for test automation, a specification must comprise the static states and the dynamic behavior of the unit under test. The specification must be formal and sufficiently detailed in order to allow for automatic processing. We call such a comprehensive and formal specification a "model." At present, such models are, in general, not available in the HMI domain. The design of the syntax and semantics of an appropriate specification language (i.e., modeling language) needs to take into account the requirements of automated testing. In order to be able to state these requirements, we need to start gaining experience with automated model-based testing, which in turn depends on the availability of models in the first place. In order to escape this chicken-or-egg situation, we have taken an intermediate step, in which the tests are performed not against the formal model of the HMI but against a prototype implementation of the HMI. This intermediate step is the topic of the next subsection.

### 1.1 Intermediate step toward test automation

The key to prototype-based test automation is that the prototype implementation is used as a stand-in for the formal HMI model, which is not yet available. Instead of testing the HMI embedded control unit (the unit under test, henceforth called the HMI ECU) against the formal model, we directly compare the states and the behavior of the HMI ECU against the states and the behavior of the prototype implementation. In the following paragraphs, we briefly describe our rapid HMI prototyping framework FLUID ("Flexible User Interface Development") and its role in the intermediate step toward model-based test automation. For a more-detailed description of FLUID, see [GES04].

FLUID is the result of our effort to close the gap that exists between the specification stage and the implementation stage in the traditional HMI development process. FLUID is an object-oriented environment that provides the modeler and the developer with the appropriate building blocks to quickly assemble prototypes of new HMIs or HMI fragments. Much of the necessary information (such as graphics, menu structure, interaction flow logic) is specified in machine-readable specification files. These declarative specifications can be used to validate the specification itself (e.g., by model checking [Kis04, BB+99]), and also to semi-automatically derive test automata and test sequences. However, the declarative specification files do not add up to a complete HMI model; some key parts of the prototypes are still programmed, which means that we do not yet have the basis for true model-based testing. However, by more and more replacing the programmed parts with formal specifications we will eventually be able to derive the prototype almost completely automatically from the specification. We will then have achieved our goal, which is to use a formal HMI specification that serves as a common basis for all phases of the development process, including automated testing.

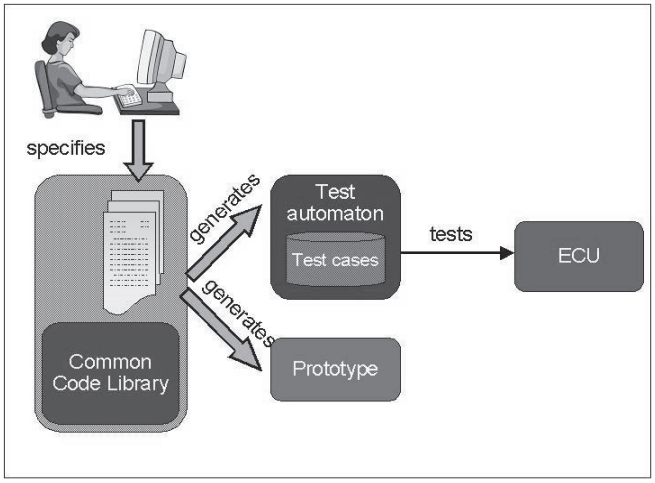
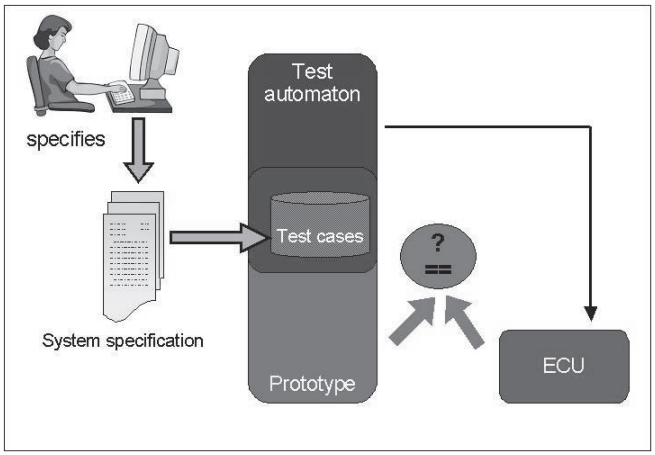



Fig. 1: long-term goal of test automation

Using FLUID in our work on test automation has several advantages. First, it allows us to have HMI prototypes stand in for formal models until such models are available. Second, by integrating tests into the prototyping environment we effectively design an appropriate test architecture, and third, we learn the requirements concerning the associated test interfaces.

Automated testing is based on triggers and observations. In our approach, a trigger is an action that is normally performed by the user of the HMI, but in the test scenario is automated by sending the bus message that corresponds to that user action. An observation is a state or an event that is observed by the test automation tool. A screenshot would represent a certain state but a bus message is considered an event. For example, such a message could be a "Play" command to the CD player. The test automation tool uses a scripting mechanism to perform triggers and to log the observations of states and events. For example, a script may contain a sequence of triggers that simulate the user's pressing of buttons. E.g., to simulate that the user shifts the I-Drive controller ("Ergo Commander") down one adds the step "`ergoCommander.moveDown();`" to the script. Moreover, the script may contain steps to

observe which bus messages are sent as a reaction by the HMI and it may take a screenshot after performing the button pressing sequence.



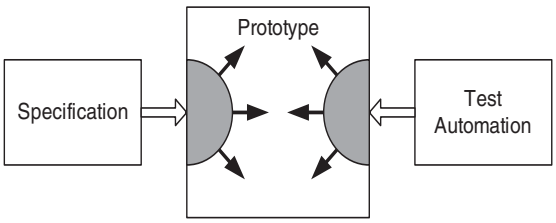

In our test scenario, the HMI ECU's test interface consists of the automotive bus systems the ECU is connected to and the screen interface. The same interfaces are also provided by the prototype built in FLUID. The availability of the same interfaces on the ECU side and on the prototype side lead to a symmetric architecture, which allows for the following

Fig. 2: Intermediate Step

important simplification. In order to perform a test procedure, we only have to execute the *same* script twice: one execution operates on the prototype and the other execution operates on the ECU. After these two runs have been completed, the test automation tool compares the two data sets that were obtained through observation and determines if the ECU complies with the reference implemented by the prototype.

### 1.2 Transition to model-based test automation

Prototype-based test automation is only an intermediate step on the way to model-based test automation. The prototype fills in for the model; it thereby acts as a mediator between the specification (which is incomplete for now) and the test automaton.




Our strategy to achieve true model-based test automation is to successively make inroads into the prototype from both sides: the specification side and the test side. Eventually, the hand-coded parts of the prototype will disappear.

Fig. 3: Testing against the prototype

On the test side, an extended interface of the prototype will allow for more useful observations than just screen shots and bus messages. This extended interface will provide structured information such as the currently focused widget, the currently displayed menu items, the current position in the menu hierarchy, the currently displayed graphics, and so on. As explained below, these structured pieces of data in the prototype will be more and more derived from the specification directly, thereby establishing a direct connection between the test automaton and the specification.

On the specification side of the prototype, we rely on progressively increased code generation. Currently, the prototype is a combination of declarative specifications and hand-programmed code. Our goal is to generate large parts of the prototype automatically from a formal specification that includes not only graphics, texts and menu structure, but also the flow logic, non-local state transitions, synchronization of various modalities (e.g., graphics and speech interfaces), and so on. (An example of a formalism that allows parts of such specifications is IML [WEA04].) In an ideal world, the prototype would be derived entirely automatically from the specification. If this were the case, a comparison of the ECU with the prototype would amount to a comparison of the ECU with the specification because the information contained in the prototype is equivalent to the information contained in the specification. In other words, in that ideal world, the prototype executes the specification. The prototype is both prototype and test automaton. Consequently the gap between specification and test has been closed.

The following table illustrates the progression from pure prototype-based testing to model-based testing. In this example, the task is to compare the text displayed in a screen segment against the corresponding text in the specification. As the specification becomes more structured and more complete, the test can pull the necessary information more and more directly out of the specification. An open issue that is shown in the "test strategy" column is the lack of appropriate test interfaces in traditional HMI ECUs. Since one cannot query the ECU for the contents that is currently being displayed on the screen, we need to perform OCR on the corresponding segment of the screen shot.

| **Step** | **Information basis (specification)** | **Test strategy** |
|---|---|---|
| 1. | No information available that the screenshot contains a text label. Only pixel data available. | Text label compared as a screenshot byte-for-byte. |
| 2. | Information that screen segment contains text label is obtained from internal prototype state (widget tree). The shown text itself is also obtained. | OCR performed on screen segment showing text label. |
| 3. | Prototype screen is generated from a specification. The composition of the screen segment and the shown text itself is contained in the specification. | OCR performed on screen segment showing text label. |
| 4. | Automated test obtains text directly and automatically from the screen specification. | OCR performed on screen segment showing text label. |

## 2 Hot Topics in Model-Based HMI Test Automation

One of the goals of the GI Automotive Software Engineering workshop is to exchange research areas, ideas and results. In this spirit, we use this space to recommend a number of focus areas to be worked on. As described in Section 1, we are moving toward true model-based test automation. This will open up a set of new questions and opportunities.

**1. Automatic selection of test cases.** An obvious starting point are random walks through the space spanned by the specification. Preference can be given to paths on which errors have been discovered in previous runs ("machine learning") and to paths that have been marked by human experts as promising ("expert teaching").

**2. Automatic analysis of test coverage.** It is clear that the traditional notion of coverage of code branches is inappropriate here. Rather, it is necessary to consider paths through the space of user events and internal and external ECU events. One challenge here is the appropriate choice of abstraction in order to link model paths with test paths. The former are sequences of partial (i.e., abstract) states. The latter are more like traces (even though they are not completely instantiated traces). For example, a test case abstracts away the exact time behavior and, instead, specifies time intervals only (if at all).

**3. Compositionality of models, compositionality of tests.** E.g., a test of the climate control unit is typically oblivious to the CD that is currently playing in the CD changer – unless one suspects an interaction between the climate control and the CD changer.

**4. Regression testing** will be enabled by the formal link between specification and test: changes to the specification will trigger re-runs of exactly those tests that depend on the changed parts of the specification. Independent tests need not (but may) be re-run.

This research agenda requires a representation that supports (1) the factorization of the model into mutually independent (or loosely coupled) components, and (2) the corresponding factorization of test sequences into independent subsequences. This compositionality property is also required of the descriptions of both states and behavior.

## Bibliography


[BB+99] Bienmüller, T.; Bohn, J.; Brinkmann, H.; Brockmeyer, U.; Damm, W.; Hungar, H.; Jansen, P.: Verification of automotive control units. In (Olderog, E.; Steffen, B., eds.) Correct System Design, LNCS 1710. Springer 1999.

[GES04] Gentner, H.; Ehrmann, M.; Salzmann, C.: Model Based Development of Automotive Human Machine Interfaces. Convergence, SAE, Detroit, Michigan, October 2004.

[Kis04] Kistler, G.: Ein model-checking-basierter Ansatz zur Prüfung von Nutzungsschnittstellen. Diplomarbeit, Technische Universität München, Fakultät für Informatik, 2004.

[ME04] Müller-Bagehl, C., Endt, P., eds.: Infotainment/Telematik im Fahrzeug. Tagungsband. Expert-Verlag, Renningen, Germany, 2004.

[UHD04] Unruh, C.; Hepp, H.; Danz, D.: Testautomatisierung f. Infotainmentsysteme. In [ME04].

[WEA04] Wegner, G.; Endt, P.; Angelski, C.: Das elektronische Lastenheft als Mittel zur Kostenreduktion bei der Entwicklung der Mensch-Maschine-Schnittstelle von Infotainment-Systemen im Fahrzeug. In [ME04].