

test methodology

Rapid Embedded System Testing Using Verification Patterns

Wei-Tek Tsai and Lian Yu, *Arizona State University*

Feng Zhu, *University of Minnesota*

Ray Paul, *Department of Defense*

A pattern approach to testing real-time embedded systems lets developers customize a set of test script templates and reuse them throughout an application's life cycle.

Testing is often difficult, and testing real-time embedded systems for mission-critical applications is particularly difficult owing to embedded design complexities and frequent requirements changes. Embedded systems usually require a series of rigorous white-box (structural), black-box (functional), module, and integration testing before developers can release them to the market. In practice, functional testing is often more important than structural testing. Similarly, integration testing is more challenging than module testing. Furthermore, functional integration testing often requires individual test scripts based on the system requirements.

This process is expensive and time consuming. A simple example illustrates the cost and effort involved to test a safety-critical embedded system. Developers often specify embedded systems using scenarios, and a typical medium-size system has hundreds of thousands of scenarios. Each scenario requires a test script, and each test script is potentially a nontrivial program that requires individual development and debugging. Assume that the average size of each test script is 2,000 LOC (line of code), and depending on the test engineer's experience level, each test script may take up to one week or two to complete. The number of test scripts will obviously be too large to be feasible or economical.

To address this problem, we propose a *verification pattern* approach to developing test scripts quickly. The VP approach classifies system scenarios into patterns. For each scenario pattern (SP), the test engineer can develop a script template to test all the scenarios that belong to the same pattern. This means that the test engineer can reuse test script templates to verify a large number of system scenarios with minimum incremental cost and effort. The VP approach also simplifies verification of system changes. For example, if the change represents a new scenario that falls within the existing SP classifications, the test engineer can reuse the existing test script template to test it. If it doesn't match any existing patterns, the test engineer needs to develop a new pattern and new test script for it.

We applied the VP approach at several industrial sites to test safety-critical implantable medical devices. The results showed significantly reduced testing cost and effort compared to existing approaches.

VP advantages

The VP approach has several advantages over conventional testing approaches:

- Most VP test scripts are developed by customizing existing test scripts, which makes

them much more consistent and reliable. We observed that 70 percent of the integration testing effort is spent on developing and debugging test scripts rather than performing actual tests.¹ Thus, reusing proven test scripts can significantly reduce debugging time and effort.

- n VPs can test timing bugs by incorporating timing constraints.
- n VPs can work both online and offline. Embedded system designers can either incorporate VPs in the system code to evaluate runtime behavior in real time or collect system behavior traces for later evaluation.

The critical issue, however, is determining the number of SPs needed to reasonably cover a large industrial application. If the application has thousands of SPs, the VP approach won't be profitable because the cost of identifying patterns, maintaining test script templates, and customizing the templates for each scenario will be too high. Fortunately and surprisingly, for many industrial applications we encountered—especially applications to real-time, safety-critical medical devices—large complex systems needed only a small number of SPs, even for complex industrial systems with hundreds of thousand scenarios. For example,

- n eight SPs covered 95 percent of system scenarios for an industrial safety-critical implantable medical device²;
- n four SPs covered 75 percent of a real-time ultra-reliable communication processor; and
- n 29 SPs covered 100 percent of a large complex industrial process-control software program.

These examples show the significant potential savings the VP approach offers.

The VP concept is related to testing concepts such as object-oriented test frameworks, test patterns, and regression testing, but the VP approach differs from them. For example, the OO test frameworks popularized by JUnit (www.junit.org), Cactus (<http://jakarta.apache.org/cactus>), and HttpUnit (<http://httpunit.sourceforge.net>) use OO design patterns such as Template Method, Strategy, and Composite.³ However, these patterns apply to the OO test framework's design. VPs apply to actual system scenarios even if an OO test framework is not used, although OO design patterns such as Template Method can be used in the VP test templates. OO test frameworks also differ from OO application frameworks: OO test frameworks organize test scripts in an OO structure,⁴⁻⁶ while OO application frameworks organize system code in an OO structure.

The test engineer can identify VPs even if the application isn't designed in an OO manner. In fact, a software engineer can develop a system in any manner, and the system behavior can still exhibit patterns. For example, one study of the property specification patterns for temporal behavior classified the bulk of specifications into a small selection of patterns.⁷ In another study, we specified these properties as scenarios and classified into SPs, and each SP corresponds to a temporal logic for model checking.⁸ Oliver Niese⁹ identified common temporal constraint patterns from libraries of test blocks and constraints and later used them in regression testing.¹⁰ The approach constructs and validates an initial test graph using either graph tracing or model-checking control.

Several researchers have proposed test patterns, but most of them codify either common knowledge, such as “Test the boundary of model,” or process patterns, such as “Test all” in a regression-testing pattern.

We identify the VPs from system scenarios, which are behavior patterns, not OO design patterns. Because the VP approach identifies patterns from system requirements only, it represents a black-box approach to testing and verification. It uses no design information to generate test scripts. The test engineer can implement the VP approach on top of an existing OO test framework. For example, we used Template Method, Factory Method, Strategy, and Composite design patterns in the test framework.^{2,11}

Furthermore, the test engineer can use VPs simultaneously with regression testing. Both

approaches emphasize test reusability. However, their approaches to reusability are completely different. Regression testing reuses existing test case scripts/cases and relies on traceability to support reusability.^{12,13} Thus, regression testing helps most when software changes are relatively minor and involve no structural changes. With significant software changes, many existing test cases might no longer apply. The VP approach, on the other hand, doesn't take system design into consideration and so can handle structural changes effectively.

Verification framework and architecture

The verification framework is a functional testing framework because, as Figure 1 shows, it is requirements-driven. It uses requirements to derive tests as well as to develop verifiers.

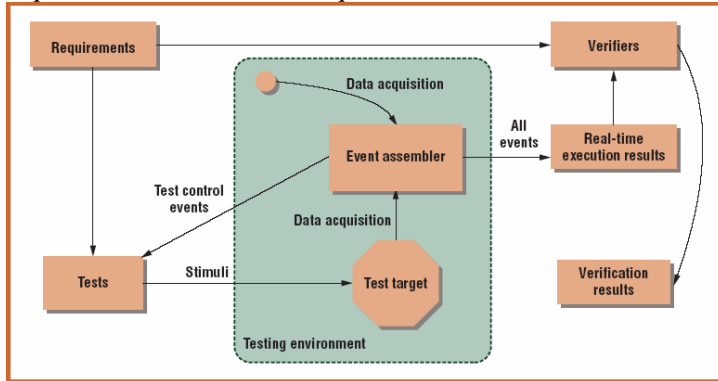


Figure 1. Verification framework.

Tests create the conditions or stimuli to which the *test target*—that is, the device or system under test—should respond as specified in the requirements. A test might consist of one or more scenarios. When a test executes, it produces stimuli and provides them to the test target. The *event assembler* records all the stimuli and test target responses, then processes these raw data to form low-level events. Sample raw data includes interrupts, timers, and register values. The event assembler further assembles low-level events into high-level events.

Verifiers continuously monitor events the system generates. A verifier takes the test target's response and checks to see if it meets the specified requirements. The verifier can operate online in real-time and offline.

Figure 2 illustrates the verification framework's structure, its components, and the relationships among them. The components within the two solid horizontal lines represent the framework's main body. The architecture considers the test target to be one of the hardware devices and treats it the same as other hardware devices.

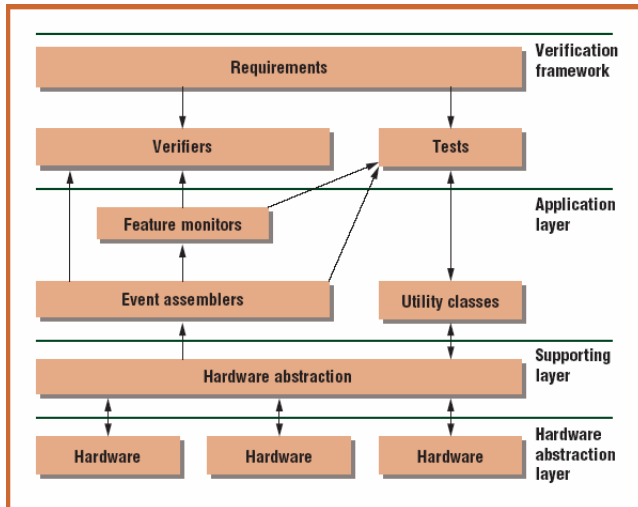


Figure 2. The verification framework's software architecture.

We can further divide the hardware abstraction layer into two sublayers: device drivers and device APIs. Normally, device drivers are system-specific and the driver mechanism provided by the operating system constrains the way they work. The device API sublayer encapsulates all device-specific and OS-specific details and provides them to the upper layers.

When a test executes, it commands the test environment through *utility classes* to produce test stimuli that are fed into the test target. The test target reacts to these stimuli by generating appropriate outputs. The test environment records all activities of both the test environment and the test target into log files. The hardware abstraction layer receives simultaneously the test environment's hardware activities and the test target activities and assembles them into hardware events.

Behavior verification ensures that the test target's behavior conforms to the specification. Upon receiving certain stimuli from the test environment, the verification framework verifies that the test target behaves as the requirement document specifies. During the behavior verification, the test system no longer interacts with the hardware environment. Instead, it takes log files recorded during test execution as input and the event assemblers assemble events using the log file data.

Normally, online behavior verification requires more computing power to process numerous events generated during real-time execution. On the other hand, offline behavior verification requires less computing resources to perform and can run on any platform.

Scenario and verification patterns

Each scenario in an embedded system development usually has preconditions (or causes), postconditions (or effects), and optional timing constraints. Our work includes a GUI-based specification tool to facilitate scenario specification.¹⁴

We define a *scenario pattern* as a specific temporal template or cause and effect relation representing a collection of requirements with similar structures. For example, a typical commercial defibrillator has hundreds of thousand scenarios. However, as Table 1 shows, just eight patterns describe 95 percent of the system scenarios.²

Table 1

Scenario patterns and requirement coverage for a commercial defibrillator

Pattern descriptor	Coverage (%)
Basic scenario pattern (BSP)	40
Key-event-driven scenario (KSP)	15
Timed key-event-driven scenario (TKSP)	5
Key-event-driven time-sliced scenario (KTSP)	7
Command-response scenario (CSP)	8
Lookback scenario (LSP)	6
Mode-switch scenario (MSP)	8
Interleaving scenario (ISP)	6
Total coverage	95

We define a *verification pattern* as a predefined mechanism that can verify a group of behavioral requirements that describe similar temporal pattern or cause and effect relations. We use the VP to verify the requirements that scenario patterns represent.

A VP description includes a class diagram (for the requirement representation) and verification state machine (representing the verification logic).

Here, we give the SP and VP for two of the eight patterns. Information about the other six patterns is available elsewhere.²

Basic scenario pattern

We describe the basic scenario pattern as follows: whenever precondition A holds, postcondition B becomes true within T period of time.

Figure 3a illustrates the pattern in a timeline; the following is an example: “Upon a warm reset, the firmware shall clear registers 7 and 9 within 80 ms.”

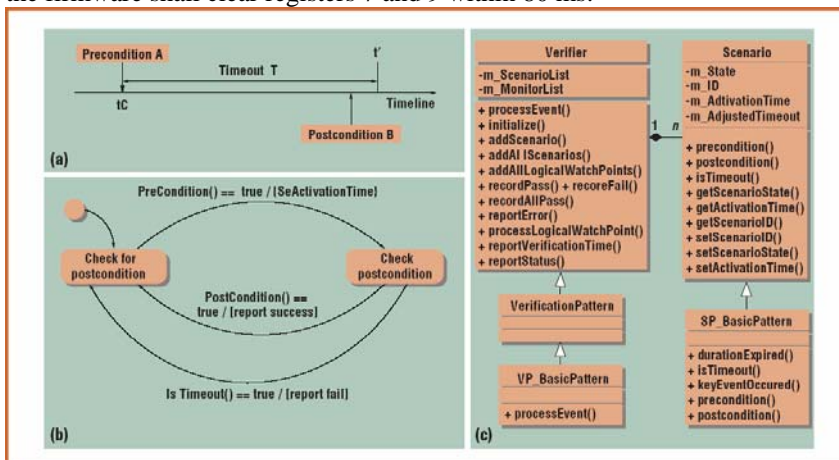


Figure 3. Basic patterns: (a) basic scenario pattern’s timeline illustration, (b) basic verification pattern’s class diagram, and (c) verification logic.

The example gives a timing requirement to verify by watching a “warm reset” event with a timer and the events of updating registers 7 and 9 within 80 ms. If the register updates come in before the deadline, the test verifies that the firmware has passed the requirement. If the test does not show the updating events before the deadline, it verifies that the firmware has failed the requirement.

Test engineers use the basic scenario pattern extensively in practice. It covers 40 percent of the

requirements in a typical implantable medical device.

Basic verification pattern

The basic VP verifies the requirements that the basic SP represents. It has a single precondition and postcondition with an optional timeout, and its use is common in real-time requirements verification.

Figure 3b shows its class diagram. Figure 3c gives the verification logic, which applies to all requirement items classified within this pattern and added to the pattern verifier during verification.

Timed key-event–driven scenario pattern

We describe this scenario pattern as follows: within the duration T after the key event, if P event, then R event is expected before t_2 .

Figure 4a illustrates this pattern, using a timeline; the following is an example: “If within the 30 seconds required to change the TachyMode to OFF, SystemMagnetApplied becomes cleared for less than 2 seconds, the firmware shall still change TachyMode to OFF after the original 30 seconds have expired.”

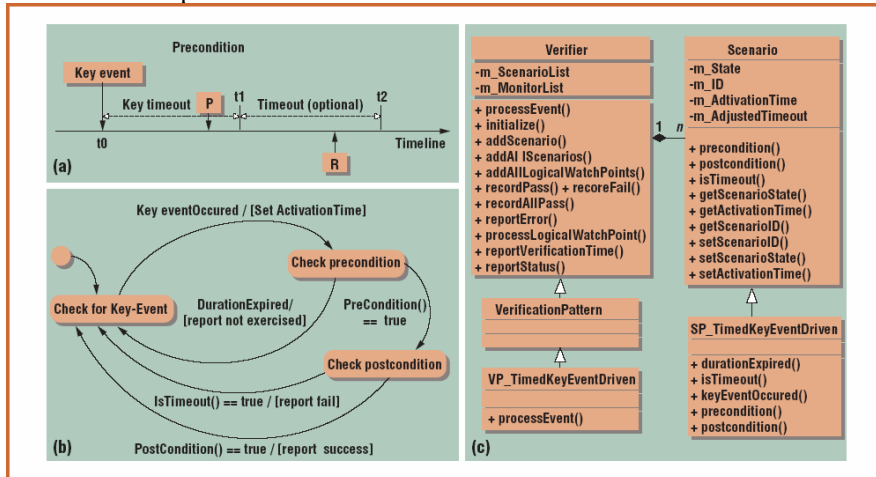


Figure 4. Timed key-event–driven patterns: (a) scenario pattern’s timeline illustration, (b) verification pattern’s class diagram, and (c) verification logic.

<<<<Production/layout note: Same as for Figure 3.>>>>

The timed key-event–driven requirement pattern requires a duration after the key event occurs. In this pattern, the precondition combines the key event and the event P that occurred during the specified duration T.

Timed key-event–driven verification patterns

This VP verifies requirements that the timed key-event–driven SP can represent. It provides an interface to decide if the duration has expired.

Figure 4b shows its class diagram. Figure 4c gives the verification logic. Unlike the basic VP process, which starts checking the precondition right away, the timed key-event–driven VP process checks the precondition within the specified duration after the key event occurs. The verifier would report “Not exercised,” if it failed to verify the precondition within the duration.

Pattern-based verification and evaluation

Table 2 shows a small embedded-system car alarm system that illustrates the VP approach. The CAS has 20 scenarios that fall within five patterns, so the system requires only five test script templates. Readers interested in the complete CAS example—including requirements, design, code, and test scripts—can visit website at: <http://asusrl.eas.asu.edu/Verification/>.

Verification process

The VP process includes four steps:

1. Collect and specify system scenario requirements. Each system scenario should specify preconditions, postconditions, and timing constraints. A tool is available to automate system scenario specification.¹¹
2. Match a scenario to a scenario pattern, if any, or create a new scenario pattern. For example, the scenarios SR5–S8 and SR11–SR20 in Table 2 share the following structure: “When conditions are true, if event *E* occurs, then take actions.” These scenarios can therefore use key-event–driven SP.
3. Generate test scripts by utilizing verification code templates. Instead of developing the test scripts line by line, the tester needs only to replace the code template parameters by actual parameters.
4. Execute test scripts and verify results.

Whenever any system requirement changes, the process repeats by first classifying the modified scenario into patterns. In many cases, the modified scenario also fits into one of existing patterns.

Because test engineers can reuse test script templates, the VP approach can potentially reduce verification costs and effort significantly, even during agile development where constant and extensive changes occur.

Evaluation of verification patterns

We used the pattern-based verification approach for an implantable medical device developed at Guidant (www.guidant.com). The VP approach reduced testing effort from 25 to 90 percent, depending on the engineers’ experience and the kind of changes.

Table 3 shows the results for various system changes including timing, action, and sequence changes. For example, a novice engineer who **was** new to the VP approach outperformed another novice using the conventional approach to a new requirement and all requirement changes. An experienced engineer learning VP achieved performance almost identical to another experienced engineer using the conventional approach in nearly all cases, and an experienced engineer familiar with VP performed best in all cases.

Table 3
Case Study of Implantable Medical Device Verification

*The numbers represent hours. For example, Group A engineering needs on average 40 hours (varies from 10 to 50 hours) to complete a new requirement item.

The test code needed to test the system offers another proof of productivity gains. Table 4 compares the use of the VP and conventional approaches for two industrial groups. Both groups have similar experience and worked on similar projects involving actual devices in an industrial environment. The project using the VP approach reduced the code size on average from 1,380 LOC per scenario to 143—almost a 90 percent reduction.

Table 4
Verification Framework Effectiveness

Characteristic //or what?//	Conventional Group	VP Group
No. of requirement items	99	114
Use of verification framework	No	Yes
Lines of code (.h) //author: What does this mean?//	60,211	6,200
Lines of code (.cpp) //Same question.//	76,412	10,100
Total LOC	136,623	16,300
LOC/requirement	1,380	143

Most embedded system companies use a product-line approach to development.¹⁵ Specifically, they usually have several product families, with the products in each family (or even in different families) sharing significant common features or structures. For example, a medical device company can have two main product lines, one for defibrillators and one for pacemakers, but these two families still share many common behaviors because, for example, a defibrillator also needs to track pace. The VP approach has significant leverage for products within a family and, because of sharing among families, can also help across product lines.

Even though we developed the VP approach for testing medical devices, the technique can also apply to other embedded applications or even nonembedded applications. The technique applies to any system that exhibits behavior patterns, and such patterns occur within telecommunications, semiconductor, manufacturing, defense, and aerospace applications (see <http://asusrl.eas.asu.edu/Verification/>).

We've extended the VP approach to cover negative aspects of testing—that is, testing the system by feeding it incorrect input.¹⁴ We're also developing a tool to automatically recognize scenario patterns from system scenarios and currently experimenting with it on several large real-time embedded applications. ■

References

1. B. Beizer, *Software Testing Techniques*, 2nd ed., Van Nostrand Reinhold, 1990.
2. F. Zhu, "A Requirement Verification Framework for Real-time Embedded Systems," doctoral dissertation, Dept. Computer Science and Eng., Univ. of Minnesota, Minneapolis, 2002; <http://asusrl.eas.asu.edu/Verification/>.
3. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
4. D.G. Firesmith, "Pattern Language for Testing Object-Oriented Software," *Object Magazine*, vol. 5, no. 9, 1996, pp. 42–45.
5. J.D. McGregor and A. Kare, "Parallel Architecture for Component Testing of Object-Oriented Software," *Proc. of Annual Software Quality Week*, Software Research Inst., 1996, <http://www.cs.clemson.edu/~johnmc/>.
6. M. Poonawala et al., "Testing Safety-Critical Systems—A Reuse-Oriented Approach," *Proc. 9th Int'l Conf. Software Eng. and Knowledge Eng. (SEKE 97)*, Knowledge Systems Inst., 1997, pp. 271–278.
7. M.B. Dwyer, G.S. Avrunin, J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification," *Proc. 21st Int'l Conf. Software Engineering (ICSE)*, IEEE CS Press, 1999, pp. 411–420.
8. W.T. Tsai et al., "Rapid Scenario-Based Simulation and Model Checking for Embedded Systems," *Proc. 7th IASTED Int'l Conf. Software Eng. and Applications*, (SEA 2003), Acta Press, 2003, pp. 568–573.

9. O. Niese et al., "Library-Based Design and Consistency Checking of System-Level Industrial Test Cases," *Proc. Int'l Conf. on Fundamental Approaches to Software Eng. (FASE 2001)*, LNCS 2029, Springer-Verlag, 2001, pp. 233–248.
10. T. Margaria et al., "Efficient Regression Testing of CTI-Systems: Testing a Complex Call-Center Solution," *Ann. Rev. of Communication*, vol. 55, Int'l Eng. Consortium, 2002, pp. xx-yy.
11. W.T. Tsai et al., "Developing Adaptive Test Frameworks for Testing State-Based Embedded Systems," *Proc. 6th World Conf. Integrated Design and Process Technology (IDPT 2002)*, CD collection, 2002.
12. A.K. Onoma et al., "Regression Testing in an Industrial Environment," *Comm. ACM*, vol. 41, no. 5, 1998, pp. 81–86.
13. W.T. Tsai et al., "Scenario-Based Functional Regression Testing," *Proc. of IEEE Computer Software and Applications Conf. (COMPSAC 2001)*, IEEE CS Press, 2001, pp. 496–501.
14. W.T. Tsai, R. Paul, L. Yu, and X. Wei, "Rapid Pattern-Oriented Scenario-Based Testing for Embedded Systems," in *Software Evolution with UML and XML*, H. Yang, ed., IDEA Group Publishing, London, , 2004, pp. 222-262.
15. D. Weiss and C.T.R. Lai, *Software Product Line Engineering: A Family Based Software Development*, Addison Wesley, 1999.

Table X. Car Alarm System Requirements

ID	Operational Scenarios*	SP**
SR1	If the alarm is turned on using the remote controller (E1), then the horn will beep once (A1) and the alarm is turned on (armed) (A2).	BPP
SR2	If the alarm is disarmed using the remote controller (E2), then the alarm is disarmed (A3).	BSP
SR3	When the driver and passengers doors, the trunk, and the hood are closed (C1, C2, C3), if the alarm is armed (E3), then the driver's door and passenger's door get locked (A4, A5) within 0.5 second (T1) and the horn will beep once (A1).	BSP
SR4	When driver and passenger doors remain unlocked (C5, C6), if within 0.5 seconds (T1) after the lock command is issued by remote controller or car key (E1), then the alarm horn will beep once (A1)	TKSP
SR5	When driver door or passenger door is open (C7 or C8), if the alarm is armed using remote controller (E1), then the alarm horn beep three times (A6).	KSP
SR6	When either of the doors is opened (C7 or C8), if ignition is turned on by car key (E3), then the horn beeps three times (A6).	KSP
SR7	When the passenger and driver doors are closed and unlocked (C18, C19), if timer passes 10 seconds (E4), then the alarm is turned on (A2) and both doors are locked (A7, A8).	KSP
SR8	When the driver door or passenger door is open (C7 or C8), if the alarm is armed using the remote controller (E1), then the horn beeps once (A1).	KSP
SR9	If the alarm is disarmed using the remote controller (E1), then the driver and passenger doors are unlocked (A9, A10).	BSP
SR10	When the driver and passenger doors are locked (C9, C10), if unlocked using the remote controller for the first time (E5), the driver door and passenger door are locked (C9, C10), then the driver door is unlocked (A9). If the event occurs for the second time (E6), then passenger door is unlocked (A10).	ISP
SR11	When the driver or passenger doors are locked (C9, C10), if the door is opened using the car key (E6), then open the door that is requested (A9 or A10) and turn off the alarm (A11).	KSP
SR12	When the trunk is closed (C11), if the trunk is opened using the car key (E7), then open the trunk (A12).	KSP

SR13	When the driver and passenger doors are closed (C12, C13), and trunk is open (C14), if the alarm is armed using remote controller (E1), then the horn beeps once (A1).	KSP
SR14	When the alarm is on (C15), if the ignition is turned on using car key (E9), then the horn beeps once (A1).	KSP
SR15	When the driver or passenger doors are unlocked (C5, C6), if the door is unlocked using car control (E8), then the horn beeps once (A1).	KSP
SR16	When the hood is opened (C16), if the door is unlocked using car control (E8), then the horn beeps once (A1).	KSP
SR17	When the trunk is opened (C17), if unlock the door using car control (E8), then the horn beeps once (A1).	KSP
SR18	When the driver door or passenger door is open (C7 or C8), if ignition is turned on using car key (E9), then the horn beeps once (A1).	KSP
SR19	When the trunk is open (C17), if ignition is turned on using car key (E9), then the horn beeps once (A1).	KSP
SR20	When the hood is open (C16), if ignition is turned on using car key (E9), then the horn beeps once (A1).	KSP

*C stands for conditions; E stands for events; and A stands for actions.

**For the acronyms, see Table 1.

Wei-Tek Tsai is a professor of Computer Science and Engineering at Arizona State University, Tempe, Arizona. His research interests include software testing, software engineering, and embedded system development. He received his PhD in computer science from the University of California at Berkeley. Contact him at Dept. of Computer Science and Eng., Arizona State Univ., Tempe, AZ 85287-8809; wei-tek.tsai@asu.edu.

Feng Zhu is a senior test engineer. His research interests include pattern-based requirements verification for real-time embedded systems; automated model-based testing methodology and process; and test classification-based, automated test-selection optimization for large-scale regression. He received his PhD in computer science and engineering from the University of Minnesota. Contact him at Guidant Corp., 4100 Hamline Ave. N., E201, St. Paul, MN 55112; feng.zhu@guidant.com.

Lian Yu is a faculty associate at the Department of Computer Science and Engineering of Arizona State University. Her research interests include software engineering, validation and verification, production scheduling, and supply chain management. She received her PhD in electrical and computer engineering from Yokohama National University, Japan. Contact her at Dept. of Computer Science and Eng., Arizona State Univ., Tempe, AZ 85287-8809; lianyu@asu.edu.

Ray Paul is a technical Director at OSD NII. His research interests include electronics engineering and software architecture, development, test, and evaluation. He has a PhD in software engineering. He is a member of the IEEE Computer Society. Contact him at OSD NII, Dept. of Defense, Washington, DC; Raymond.Paul@osd.mil.