

Verification Patterns for Rapid Embedded System Verification

W.T. Tsai, F. Zhu‡, L. Yu, R. Paul*, C. Fan

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287

wtsai@asu.edu Tel: (480)727-6921

‡Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455

*Department of Defense
Washington, DC

ABSTRACT

Test result verification is always a costly task for embedded system testing. This paper presents a systematic process to develop verification patterns and use these patterns to verify test results for state-based real-time/embedded systems. The verification patterns are organized into an object-oriented verification framework so that it can be adaptive to changes rapidly. The verification patterns are reusable and thus can save significant time and effort in constructing the test execution infrastructure and generating test scripts. This paper describes a systematic process to perform rapid embedded system verification: 1) abstracts verification patterns based on requirement/scenario patterns analysis; 2) generates test cases/scripts from the verification patterns; 3) develops data acquisition component for raw data retrieval and event assemblers; 4) executes test scripts locally or remotely, and 5) collects and analyzes the test results. This process has been applied to a car-alarm system and implantable device applications, which shows the framework developed can perform the verification tasks efficiently.

1. Introduction

Developing high-confidence real-time embedded systems is expensive and costly, and one of the major difficulties is verification and validation (V&V) of system behavior. Even though significant research has been performed on formal specifications and model checking, most of V&V activities still focus on testing as the primary quality assurance technique. The cost of testing can be as high as 50% to 70% of total development effort and budget in various industries such as aerospace, implantable medical devices, telecommunication, and defense.

Testing is often difficult, but testing real-time embedded systems for mission-critical applications such as aerospace and medical devices is particularly difficult due to the complexity of embedded system design and functionalities; and frequent requirement changes. Real-time embedded systems often have strict timing requirements and they must be observed otherwise they can have a serious impact to the user. For example, an implantable defibrillator must wait for a certain period after detecting a heart problem before it performs its therapy. The waiting is necessary because the capacitor must be sufficiently charged before a therapy can be applied. If the therapy is applied before the capacitor is fully charged, the therapy may not be effective for the patient at risk. Timing constraints can become complex due to interaction of thread scheduling, process synchronization, and external user/environment interaction.

Conventionally, requirement verification has often been done in a case-by-case approach [2]. For each requirement testers implement a procedure that observes the pre-conditions of the requirement and verifies its post-conditions. Then testers put all the verification procedures together to form a regression suite that would be run again and again against the changes or bug fixes of a product. It is true that for each requirement tester only implements its verification procedure once. However, the approach also raises the following issues:

- Each new requirement requires a new procedure implementation, no matter how similar it is to some of the existing requirements.
- If a requirement gets changed, its verification procedure has to be changed too.

- Separately implemented verification procedures need to be debugged separately, which means higher cost for development.
- Separately implemented verification procedures are often implemented by different people and normally in different coding styles, which often result in higher cost of maintenance.

This paper addresses these issues by devising a structured approach for rapid testing of real-time embedded systems. The principal technique used in the proposal is *verification pattern*, with which testing activities are classified and organized. Thus, it will enable systematic and rigorous analysis based on specifications, and facilitate adaptive approach to reveal and check the timed program behavior of real-time embedded systems, such as safety-critical medical devices and communication processors.

The verification pattern approach to rapidly test embedded systems has many advantages over conventional testing approaches, namely:

- Most of test scripts used to verify the system are developed by customizing existing test scripts, and thus the test scripts are much more consistent and reliable. It is estimated that 70% of effort during the integration testing is spent on debugging test scripts rather than developing the test scripts and performing actual tests. By reusing the proven test scripts, the debugging time and effort are significantly reduced.
- The verification patterns can be used to test timing bugs as timing constraints can be incorporated into the verification patterns.
- The verification patterns used state-based specifications, and thus they can be easily used with state-based specifications such as SDL, Statecharts and UML for embedded system development. SDL is commonly used in telecommunication, and statecharts for state-based mission-critical applications, and UML for working with OO designs and implementations with states.
- The verification patterns can be used in both online and off-line basis. The verification patterns can be incorporated into the system code to evaluate the system behavior at runtime in real time, or it can be used to evaluate the system behavior after the traces of the system behaviors have been collected.
- The verification pattern approach is a practical approach to test real-time reactive systems with proven success in a medical device producer. Many V&V and testing techniques have been proposed in

the literature, but few have been tried in industrial practices in verifying large industrial products, even fewer techniques have been shown to be successful.

The paper is organized as follows: Section 2 illustrates verification patterns and verification framework. Section 3 shows the verification process. Section 4 further explores verification patterns with concurrent scenarios. Section 5 discusses the benefits of applying verification patterns. Finally, Section 6 concludes the paper.

2. Verification Patterns and Verification Framework

A verification pattern is techniques similar to object-oriented (OO) design patterns [5] but applied to verification. Verification patterns are compatible with modern OO development techniques, because this research organizes verification patterns into OO framework – Verification Framework using a variety of design patterns.

2.1 Verification Patterns

The behavioral specification for real-time systems often uses a pair of pre-condition and post-condition to represent cause-effect relations or logical event sequences. An example of timing requirement is:

“Upon warm reset, the firmware shall clear register x and y within 80ms”.

where “warm reset” is the cause and the “clear register x and y” is the effect. This kind of requirement statement is referred as basic requirement pattern.

A survey on several firmware requirement specification documents produced a statistics as shown in Table 1 [13]. Here, the requirements are classified into several requirement patterns. It is observed that a large portion (around 40%) of the behavioral specification was represented as basic requirement pattern, while the rest could be classified into a handful other requirement patterns.

Pattern	Coverage(%)
Basic requirement pattern	40
Key-event driven requirement pattern	15
Timed key-event driven requirement pattern	5
Key-event driven time-sliced requirement pattern	7
Command-response requirement pattern	8
Lookback requirement pattern	6
Mode-switch requirement pattern	8
Interleaving requirement pattern	6
Total	95

Table 1 Scenario Patterns and their Percentage in an implantable Medical Device

Defining requirement patterns is not the ultimate goal. Requirement patterns are used to classify requirements into groups that can be verified categorically. By providing unique and same verification mechanism for each requirement pattern, the requirement verification process is systematically simplified in the context of the verification framework. This paper defines verification patterns as follows:

Definition: A verification pattern is a pre-defined verification mechanism that can be used to verify a group of behavioral requirements that describe similar temporal pattern or cause-effect relations.

The following presents an example of verification patterns corresponding to the requirement patterns listed in the previous table that are most commonly seen in the requirement specification of real-time embedded systems. Figure 1 shows Timed Key-Event Driven Requirement Pattern that describes two timing constraints for three events:

Within the duration $t1$ after the key event, if P event then R event is expected before $t2$.

As an typical example in an implantable defibrillator, when the device detects a heart problem, the capacity must be charged before it can apply a therapy to the patience, and this scenario shows three events (detection, capacity charged, and therapy applied), and timing constraints between these three events.

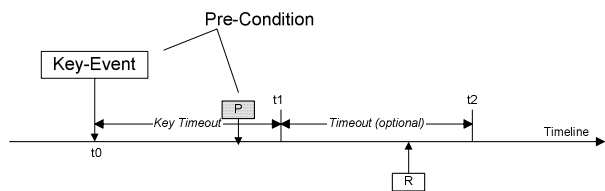


Figure 1. Timed Key-Event Driven Requirement Pattern

The timed key-event-driven verification pattern is used to verify requirements that can be represented using the timed key-event-driven requirement pattern. It provides an interface to decide if the duration is expired (Figure 2 class diagram). Figure 3 shows the state model for Timed Key-Event Driven Verification pattern. This pattern is so popular that if the medical device has 7,000 system scenarios, 15% of system scenarios belong to this pattern, i.e. 1,050 scenarios. Furthermore, each system scenario belonging to this pattern can be verified using the same verification pattern as shown in Figure 3. The verification pattern starts checking the pre-condition right away, the verification process here checks the pre-condition within the Duration after the key-event occurs. The verifier

could report “not exercised” if it failed to verify the pre-condition within the duration.

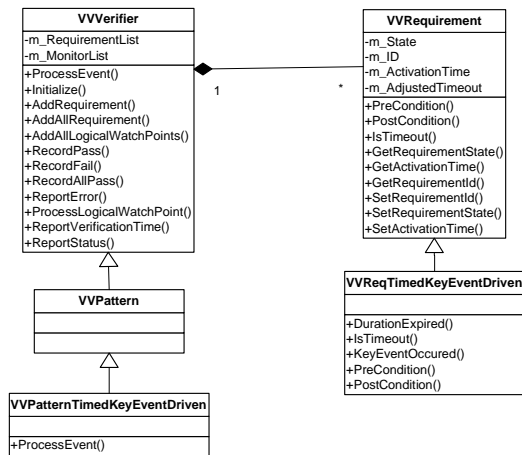


Figure 2. Class diagram

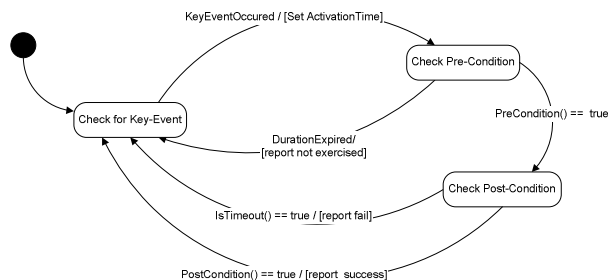


Figure 3. Timed Key-Event Driven Verification Pattern

2.2 Verification Framework

The foundations for the framework are: Requirement, Test and Verifier. The relation and the interaction among these three core concepts form the main skeleton of the framework. *Requirements* are the sources of verification and validation. The main task of the framework is to verify that the test target’s behaviors meet the requirement specification. *Tests* {xe "scenarios"} create the conditions or stimuli to which the test target, the device or system that is under test, should respond, as specified in the requirement. *Verifiers* continuously {xe "verifiers"} monitor the events generated by the scenario. A verifier takes the response of the test target and check to see if it behaves as specified by the requirement.

The verification patterns can be implemented using an OO design such as shown in Figure 4. The design employs a verification engine is the main driver of the whole test system. Test engine normally includes several parallel tasks such as Scenario Driver, Event Dispatcher, Event Assembler Driver, and Verifier Driver. The raw

The scenario patterns are mapped to verification patterns using state machine model as explained in the previous section. Once a scenario is specified, it is necessary to compare this new scenario with the existing scenario patterns to see how it can be classified into one of existing patterns, or a new scenario pattern is necessary. This process can be simply represented in Figure 7.

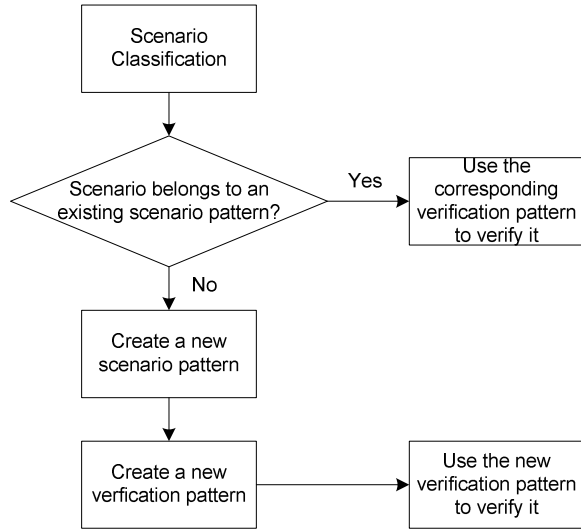


Figure 7. Process of verifying using verification patterns

This paper takes a car-alarm system as an example, and four scenario patterns are identified from the systems requirements:

- *Basic* requirement pattern: the requirement items are “alarm horn can be turned off by the remote controller”, “after the alarm is turned on, the horn will give one beep”.
- *Timed key-event* requirement pattern: the requirement item is “If both doors have been closed but unlocked for 10s, the doors should be locked and alarm should be turned on in one second”.
- *Key-Event Driven Time-Sliced* requirement pattern: the requirement item is “the alarm system is disarmed by the remote controller 0.5 second after both driver’s and front passenger’s doors are unlocked”.
- *Interleaving* requirement pattern: Interleaving requirement pattern allow tests to verify requirements that have pre-conditions and post-conditions interleaved with each other (Figure 10).

Figure 8 and Figure 9 show key-event driven time-sliced scenario pattern and key-event driven time-sliced verification pattern respectively.

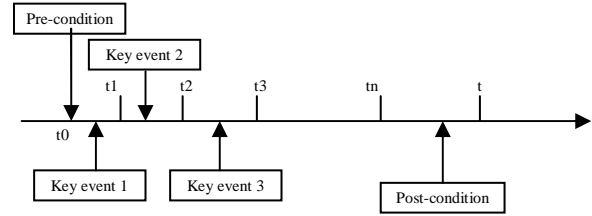


Figure 8. Key-event Driven Time-sliced Scenario Pattern

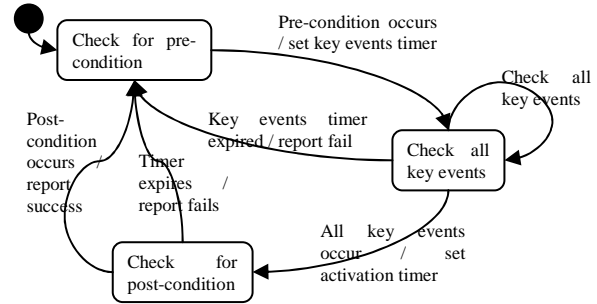


Figure 9. Key-event Driven Time-sliced Verification Pattern

The interleaving pattern is dynamic in that at least one of the post-condition is not pre-determined. We could expect to verify dynamically determined post-conditions after its pre-condition occurred, depending on the occurrence of another requirement. This requirement pattern allows categorizing requirements that have randomness in the occurring of their pre-conditions, and have sharing resources.

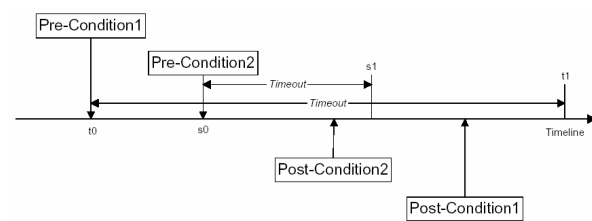


Figure 10. Interleaving Requirement Pattern

3.3 Generate Test Cases/Scripts from Verification Patterns

Scenarios describe the system behaviors responding to stimuli under different conditions. Test execution however needs one and only one execution path of a scenario. A thin thread describes the single execution path of the system, and each path in a scenario corresponds to one thin thread [6]. Thus thin threads can be obtained by parsing scenarios. Each thin thread contains *pre-conditions*, *events*, *actions*, and *post-conditions*, which are mapped to *setup*, *execution*, and

verification parts in an Execution/Verification Template (EVT) by following the steps:

- Map pre-conditions and events in a thin-thread to the *setup()* of the EVT;
- Map the sequence of actions in a thin-thread to the *execution()* of the EVT; and
- Map the post-condition in a thin-thread to the *verify()* of the EVT.

The EVT is designed and implemented using the Template Method pattern that allows the tester to customize thin threads. The tool supports automatically generating thin threads from scenario, and exports them as XML files.

Once thin threads are available, the tool generates test cases and test scripts for test execution. First, the tester needs to supply data related to the input of thin threads, and then the tool generates test inputs based on different testing techniques:

- Random testing: the tool will take random sample data from the data set.
- Partition testing: the tool will select sample data from each partition of the data in the data set.
- Boundary value testing: the tool will take the boundary values of the partitions of data in the data set.

3.4 Develop Data Acquisition Component

To verify system behaviors and functions, it is necessary to collect all related data information. Data Acquisition Component (DAC) is a hardware device or software entity whose responsibility is dumping raw data from the system while it is running. There are two different approaches for raw data retrieval: intrusive and non-intrusive. Intrusive approach need tester to modify the original system to embed the data acquisition component inside. For non-real-time system, this approach may not affect too much, while for the real-time system, especially those systems that have hard time constraints, intrusive is problematic. The system's performance will be degraded since the system has to do extra jobs. The DAC embedded in the original system will increase the response delay and add the system pay-load. On the contrary, non-intrusive approach will not bring negative effects to the test target, but it consumes extra cost of hardware and software to collecting raw data from outside of the system. It imposes higher requirements to the test environment for the ability to know adequate information about the system. Choosing intrusive and non-intrusive depends on the situations. In the car-alarm system, intrusive approach is applied.

3.5 Execution of Verification

Figure 11 shows the architecture to perform embedded systems verification [9]. The architecture of the proposed system has 3 layers on the host machine: 1) specification layer, 2) database layer, and 3) interaction layer. The specification layer creates scenario specifications, identifies scenario/verification patterns, and generates test scripts. Using scenarios information, specification layer perform a variety of analysis: such as dependency analysis, completeness and consistency check, redundancy check, and risk analysis [10]. The database layer stores the patterns, test cases/scripts and test history. The interaction layer behaves as a communicating agent with remote systems.

A test master performs the following functions:

- Schedules test scripts, and initiates the test by sending test scripts to agents either locally or remotely using TCP/IP or SNMP (Simple Network Management Protocol) [12][10][8];
- Collects the test result data from agents, and store into data base.

While an agent provides the following functions:

- Upon receiving commands from the test mater, it executes the test by first setting the SUT to the state specified by the precondition in the test script, generating events or actions according to script, and verifying results returned by the SUT by comparing the actual results with expected results;
- The agent can also act as a proxy to collect and trace system performance data such as timing and failure data under various loads. Such data will be useful for timing analysis and reliability estimation by the host.
- Sends results information to the master.

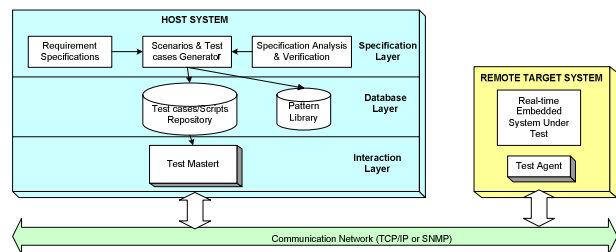


Figure 11. Verification Execution Architecture

4. Benefits of Verification Patterns

The verification pattern approach to rapidly test embedded systems has many advantages over conventional verification approaches, namely:

- A complex system can be rapidly tested by using few verification patterns. This leads to a significant reduction in the cost and effort of verification.

- Testers can take advantage of verification framework to handle requirement changes.
- The verification patterns can be used to test timing bugs once timing constraints are incorporated into the verification patterns.
- The verification patterns can be applied in both online and off-line basis.
- The verification patterns, like OO design patterns, can be catalogued and used in different projects even in different application domains.

This pattern-based approach has been used at Guidant (second largest medical device companies in the U.S.) with significant results. It reduced testing effort from 25% to 90% depending on the on experience level of engineers involved and the kind of changes (Table 2).

		Experienced			Not experienced		
Data contributor: Process X: without V-Framework Process Y: with V-Framework		Group A	Group B	Group C	Group D	Group E	
		Experienced in existing testing system	Experienced in the verification framework	Experienced in existing testing system and migrating to the	Learning existing testing system	Learning the verification framework	
Process used		Process X	Process Y	Process Y	Process X	Process Y	
Learning curve	Domain knowledge	Proficient	Proficient	Proficient	Novice	Novice	
	Testing Techniques	Proficient	Proficient	Novice	Novice	Novice	
Productivity	New Requirement	40 (10-50)	10 (4-50)	16(4-50)	90(80-100)	66(50-100)	
	Requirement Changes	Timing Change	6(2-10)	4(2-8)	6(2-10)	20(8-40)	14(8-20)
		Action Change	30(10-50)	20(4-50)	25(4-50)	54(30-100)	36(20-100)
		Sequence Change	27(10-40)	8(4-20)	10(4-20)	42(20-60)	20(10-40)

Table 2 Scenario Patterns and their Percentage in an implantable Medical Device

Another evidence of productivity gain is in terms of test code needed to test the same system -- this approach has reduced on average the code size from 1380 LOC/scenario to 143, or about 89.6% reduction (Table 3). Furthermore, the test code developed without a pattern may vary from one scenario to another depending on the engineers involved, while the design using verification patterns will be consistent because the same verification code test all the scenarios belong to the same scenario pattern. The consistency in the test code will save significant time in debugging, because it is observed that 70% of system testing effort is actually spent on debugging test code for medical devices [7].

	Feature Group 1 (BdyPCNG)	Feature Group 2 (DiagACPT)
No. of Requirement Items	99	114
Use Verification Framework	No	Yes
Lines of Code (.h)	60211	6200
Lines of Code (.cpp)	76412	10100
Total Lines of Code	136623	16300
LOC / Requirement Item	1380	143

Table 3 Verification Framework Effectives

5. Conclusion

This paper proposes an adaptive system verification process using verification patterns that facilitates rapidly testing state-based real-time embedded systems. 1) Develop scenarios from the system requirements; 2) Identify the potential requirement patterns and their associated scenarios, and map to verification patterns; 3) Generate test cases/scripts from verification patterns; 4) Develop data acquisition component for raw data retrieval and event assemblers; and 5) Execute test scripts and collect results. By using this verification framework, testers can perform numerous verifications efficiently and quickly at minimum cost whenever the system or its requirement changes. Interleaving scenario pattern is also discussed, which can be used to verify the interaction among scenarios that share common information. Verification patterns have been used to perform requirement verification for some commercial embedded systems in companies, and bring high productivity.

Reference:

- [1] X. Bai, W. T. Tsai, R. Paul, K. Feng, and L. Yu, "Scenario-Based Modeling and Its Applications to Object-Oriented Analysis, Design, and Testing", Proc. of IEEE WORDS 2002, pp. 140-151.
- [2] B. Beizer, *Software Testing Techniques (2nd ed.)*, Van Nostrand Reinhold Co., New York, NY, 1990
- [3] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.
- [4] M. Fayad, D. C. Schmidt, and R. E. Johnson, *Building Application Frameworks*, Wiley, New York, NY, 1999.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Paterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, MA, 1994.
- [6] R. Paul, "End-to-End Integration Testing: Evaluating Software Quality in a Complex System", Proc. of Assurance System Conference, Tokyo, Japan, 2001, pp. 1-12.
- [7] W. T. Tsai, X. Bai, R. Paul, and L. Yu, "Scenario-Based Functional Regression Testing", Proc. of IEEE COMPSAC, 2001, pp. 496-501.
- [8] W. T. Tsai, L. Yu, A. Saimi, and R. Paul, "Scenario-Based Object-Oriented Test Frameworks for Testing Distributed Systems", to appear in Proc. of IEEE Future Trend of Distributed Computing Systems, 2003.
- [9] W. T. Tsai, R. Paul, L. Yu, A. Saimi, and B. Xiao, "Verification of Web Services Using an Enhanced UDDI Server", to appear in Proc. of IEEE WORDS, 2003.
- [10] W. T. Tsai, L. Yu, X. Liu, A. Saimi, and Y. Xiao, "Scenario-Based Test Generation Tool for Embedded Systems", to appear in Proc. of IEEE IPCCC 2003.
- [11] W. T. Tsai, Y. Na, R. Paul, and F. Lu, "Adaptive Scenario-Based Object-Oriented Test frameworks for Testing Embedded Systems", Proc. of IEEE COMPSAC, 2002, pp. 321-326.
- [12] W. T. Tsai, L. Yu, R. Paul, T. Liu, and A. Saimi, "Developing Adaptive Test Frameworks for Testing State-based Embedded Systems", Proc. of IDPT, 2002.
- [13] F. Zhu, "A Requirement Verification Framework for Real-time Embedded Systems", PhD dissertation, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, 2002.