CHAPTER III

Architectural Frameworks and Models for test case Generation
3.0 Architectural Frameworks and models for Test Case Generation

3.1 Introduction

Testing of embedded systems is complex. Testing of embedded systems must be done quite early in the development of the system. Development of the embedded systems involves analysis, design and development of both Hardware and software in individual paths by using "V" model. The testing of Hardware and Software is done independently and they are tested once again after integration.

Testing of the software is to be carried considering the whole code instead of individual units to guarantee proper running of the software. The testing strategy must include various locations of testing, and also different methods used for the testing.

Comprehensive testing of the embedded systems involves conducting the Unit Testing, Integration testing, END-TO-END testing, and Regression testing. The testing process should consider all aspects of testing Hardware, software and both.

Embedded systems are event driven and the events occur in some patterns. The occurrence of patterns of events has to be considered to confirm that the embedded system is bug free and that the event processing is done exactly as per the application requirement.

When it comes to embedded systems the test scenarios basically should be grouped into Host based, Target Based and Target based along with the Host. The comprehensive testing requirements of an embedded system have been covered in the chapter 2.0. The comprehensive testing of the embedded systems involves the use of HOST, Target and the HOST + Target Location. Just testing the software for its reliability alone is not sufficient.
Testing the Hardware and Testing of the Hardware along with Software is equally important. A complete discussion on the kind of test cases that can be executed at each of the locations using several of the methods has been discussed in the Chapter-2.

Testing embedded systems is a challenge. Users are intolerable of buggy embedded systems. Embedded systems are low cost solutions and generally do not support the resources that are required to conduct inline testing of the embedded systems. Embedded systems are built for a specific application and generally involve Requirements gathering, analysis, design and testing of both hardware and software and integration of both embedded systems are event driven and the events are to be processed and control actions are to be taken in real time.

Testing involves test case generation, test execution and recording of the test result and conducting audit trail to find the bugs and verifying the completion of comprehensive testing of the embedded systems.

Several Frameworks are proposed in literature [Bahsoon 2003], [Christo A 2004], [Dobrita L 2002], [Duenas JC 1998], [Folmer E 2003], [Hsiung 1998], [Kun T 1995], [Lung T H 1997], [See wv 2000], [Tsai WT 2003], [Zhu F 1998] for evaluating the Framework architectures, Software component Development, Analysis of test Methods, Architectural assessments, Application development Frameworks, Identification of test scenarios, and Analysis of Frameworks. As such no architectural frameworks for integrated testing of the embedded system are existent and therefore, it is necessary that an investigation is made in this regard.

Several models are in use for conducting the analysis, design, and implementation of the loaded systems. Literature survey reveals that theses models are still being used for developing the embedded systems. The models include use cases, Behavioral models such as Sequence diagrams,
activity diagrams, collaboration diagrams etc. These Models are based on state based systems and do not consider the event occurrence patterns that happen within embedded systems.

Genetic algorithms [Jin-Cherng 2000], [Sthaner 1998], [Wegner 1997] are also used for test case generation which are based on identification, selection and testing of a path. Algorithms are based on fitness functions. Theses algorithms are also state based and do not consider event model of the embedded systems.

Test Selection is the most important issue whether it is functional testing, integration testing or regression testing. Many techniques have been introduced by several authors that include test-tube, scenario based Functional regression testing, end-to-end testing, pattern based testing etc.

Semantic models help developing test systems using which the software testing and maintenance can be carried. The semantic models help collecting the data from different life cycle stages of development and transforming the data into test cases, test scenarios and test processes and also defines a model for software maintenance.

Several of the semantic models are in existence in the literature for converting the design specifications into Test cases, symbolic model checking specifications, coupling outputs derived from analysis and design of the applications, [Bromstrut 1989], [Buhr 1998], [Dammag 1999], [Drusinsky 1985], [Henzinger 1992], [Kosters 2001], [Probert 2001], [Spence 1994]. However, these models have not considered the analysis and design outputs which are event and pattern related. These models also have not included the components that are required for conducting the actual testing of embedded applications. The test cases that could be generated by these models do not take into consideration of the need to conduct the END-TO-END testing.
A Semantic model is presented in the section 3.3 that identifies all the models required to generate the test cases and also undertake the testing of the embedded systems.

Identification of test scenarios from the Requirements specification is the Key issue. [Richard 1998] recommended a hierarchical structure of representing the scenarios. The scenarios are hierarchically organized for better understanding and deriving of the actual scenarios. The hierarchical organization reflects the functional dependencies between the scenarios and the composition/decomposition relationships between the scenarios.

Scenarios and use cases are used widely for the design and development of the software. Scenarios can also be used for testing the software [Rayadurgam 2001], [Wei-Tek 2001], [Chen 1994], [Bai X 2001], [Bengtshon 1998], [Guan R 2002], [Hsia P 1994], [Joiner K 1994], [Kim YG 1999], [Kirani S 1994-2], [Koch B, 1998], [Liu M 2002], [Miga A 2001], [Offutt J 1999], [Offet J 2000], [RRregnell 1998] [Tsai 2004], [Tsai 1999].

In case of embedded systems the test scenarios basically should be derived from the functional requirements of both Hardware and software. The scenarios identified are helpful for testing the Hardware, software and the Software along with hardware when comprehensive testing is to be carried.

The comprehensive testing of embedded system requires that the testing be carried on the HOST and on the Target and the Target along with the HOST. Process models help understand the process involved in undertaking the testing. In section 3.4, comprehensive coverage of process models that can be used for undertaking testing is presented.

Embedded systems require rigorous testing as any defects which may cause failure of the entire system as whole. Detecting the reason for the
defect and rectifying the defect is not possible as it is not easy to simulate the occurrence of the defect again either offline or online. The testing of the embedded systems must be done in a faster manner so that cost of testing is reduced. Comprehensive Testing of embedded systems requires that testing of hardware, software and the integration of both must be carried. Different kinds of testing should also be carried which include module testing, functional testing, integration testing, regression testing and component testing to ensure that the embedded systems are designed and developed properly and that all eventual events are handled properly. [Bai 2002] [xiaoying. B 2002] have proposed scenarios based modeling to identify all the scenarios required to generate test cases which are necessary to carry all different types of testing.

Scenario based testing has been proposed using which systematic testing can be carried including the test case generation, test coverage analysis, and carrying the regression testing. It is necessary to construct a data model using which all kinds of testing of embedded systems can be carried. In section 3.5 scenarios based test case generation and test execution is presented. The model suggested will help automating the process of test case generation, conducting coverage analysis, traceability analysis and ripple effect analysis. The data models also help conducting comprehensive and complete testing in a very faster manner and help reducing the cost of testing of embedded systems as low as possible.

Embedded systems are event driven. The events are either system driven or initiated by an external environment through sensing devices such as Temperature sensors. The sensed data is transmitted by a production system and captured by an embedded system. The inputs are processed and the processed results are used to trigger signals that control the physical environment. The entire processing can be represented in terms of a physical thread of execution.
An event process is a functional requirement and the devices involved needs a function be executed as per certain timing considerations and the timing are to be achieved by following a particular pattern.

[Buschbusthmann 1996], [Buschbusthmann 1997], [Douglash 1999], [Fowler 1996], [Fire smith 1996], [Gross 2001], [Kent 1994], [Konrad 2003], and [Robert 2004], have presented different types of patterns which include Software Architecture patterns, Design patterns, Pattern Languages for testing the object oriented systems, Analysis Patterns, Non Functional requirement Patterns and , Requirement Patterns.

[Tsai 2003] has presented Patterns for verification of the embedded systems which provide the common testing process for a defined architecture. However, the relationships of the patterns with END-TO-END testing of the embedded system require attention as they provide the Pre and Post conditions for testing the embedded systems.

Integration testing of embedded systems, if done using the END-TO-END testing, the event based processing that meets timing consideration can be aptly be undertaken. Thin threads provide the minimal basis for processing an event. [Bai X 2004], [Tsai 2001] have proposed the integration testing of the loaded systems based on END-TO-END testing which is the minimal of integration testing required to be carried. The testing models proposed by them are extended in this thesis to conduct END-TO-END testing of the embedded systems. Section 3.6 deals with the architectural models using which END-TO-END Testing can be undertaken.

In the case of Embedded Systems the changes do take place both in hardware and software. The changes carried at one location may affect other parts of the system. All the affected parts of the system due to the changes carried must be tested. Embedded systems also stand to change and therefore need to be tested.
In Literature [Bai X 2001-1], [Baradhi 1997], [Bates 1993], [Joiner 1993], [Pfetschner 2003], [Rothermel 1996], [Tip F 1995], [Tsai 2001], [Von Mayrhauser 1994], [wang Y 1996], [Yau SS 1978], [Zhu F 1998], [Zulkernine 1992] [Huang 1996] have presented several models for conducting distributed END-TO-END Testing Management, Comparative study of regression testing Algorithms, the process of incremental program testing using dependency graphs, Analysis of Ripple effect, program slicing and dependency, Analysis of Regression testing Techniques, Functional Regression testing, Domain Based Regression testing, and Automating the regression testing. However, the regression testing of the embedded systems needs to consider the event driven model and the process that is related to the event processing must be tested when a change is initiated to any of the elements that is part of the processing of the event.

Several methods are proposed in section 3.7 for undertaking the regression testing of the embedded systems.

### 3.2 Architectural Framework for Building Test Environment

An Architectural Framework for comprehensive testing of the embedded system is necessary. In this section the same is presented. Fig 3.1 shows the testing environment required for carrying the comprehensive testing of the embedded systems.

User interface in the HOST will not only interact with the End user but also will interact with all the testing methods proposed in the HOST. The user interface at the HOST will interface with the Communication modules at the Target and the Logical Analyzer for initiating the required testing and obtaining the test results and storing the same at the HOST for conducting the test analysis subsequently.
The way a test case is used is dependent on whether testing is done at HOST or Target or at both at Target and HOST. Comprehensive testing of the embedded system involves carrying testing at different locations and using several methods.

Different test cases scenarios, testing patterns are to be identified in the beginning which are simple and complex and are sufficient enough to carry all types of testing.

A test scenario and test pattern is tested by testing a set of test cases. A test case is tested at a location by using one or more of the methods which include testing through Scaffolding, Instruction set simulators, Assert Macros, Logic Analyzers or In-circuit Emulators or software only monitors.

Most of the software which is hardware independent is tested on the HOST and the Host process must provide the facility for scaffolding and
insertion of the Assert macros in the source code and also should support testing of the Hardware independent code. HOST provides a user interface to initiate a testing process and also automates the testing using the test scripts. The test cases and test results are stored on the secondary devices, using which the coverage analysis, test completions, audit trails etc. can be conducted.

The host will also provide interfacing with instruction set simulator to carry some specific types of testing. The HOST provides interface with the Target to test Hardware dependent code. Some types of testing which is device based can be carried by using both the target and the HOST. The online testing of the Hardware and software requires that the ES application adaptable for a change and testing is carried every time a bug is traced or the existing code is modified or when new code is added to the system. The test gadgets required for testing an embedded system must be interfaced with the HOST based software so that testing of the hardware is undertaken under the commands issued from the HOST.

The mapping of a test environment to carry testing at different locations and test methods is shown in the Fig 3.2. The classification of test scenarios as atomic and complex will provide the basis for recognizing the test scenarios to be the best cases for conducting integration testing and regression testing.

3.3 Architectural Framework for a Semantic Model

Semantic Architectural Frameworks help in modeling the testing processes and environment from for various outputs obtained through various life cycle stages of development which include Requirements, gathering, Analysis, Design and development.

[D. Deng et al, 2004] have proposed a semantic model for software testing and maintenance. The model is based on the collection of data at
different life cycle stages of development and transforming the data into test cases, test scenarios and test processes and also deals with software maintenance.

Fig 3.2 Test environment for testing an embedded system

Developing high quality fault tolerant embedded systems is one of the main objectives of software and hardware engineering of the embedded systems. Comprehensive testing of the embedded systems considering Hardware, Software and both is very much important as any failure of the embedded system will move the entire system to a halt causing severe obstruction to the production system which is monitored and controlled by an embedded system. The failures of embedded systems which are connected to the Mission Critical and safety critical systems will at times lead to disaster. Some production systems which are either mission critical or safety critical cannot be shutdown or made offline by isolating the embedded system when
a fault occurs. The faults are to be diagnosed online and corrective actions are to be taken. Even the maintenance of the software for either correcting the software or adding more functionality must be done online.

Testing of the embedded system must be done online after the embedded system is connected to the production system. Any amount of testing done offline will not guarantee the full proof working of the embedded systems as it is not possible to simulate the occurrence of uncommon errors that may occur in the production systems.

In the past several of the techniques have been invented to undertake testing of hardware and software and the techniques are put into practice and people who are using these techniques are not quite sure of the effectiveness of those techniques as the techniques are loosely coupled and are not very comprehensive.

Several Models based testing techniques have been invented in the past which are suitable for state based systems. These techniques have been used to specify and run tests and detect the defects with the help of the test results by visualizing the failure points within the model. These models also provided with the components that check the consistency between requirements and the system constructed.

Some of the model based testing techniques are based on Formal or Semi formal descriptive languages which include Extended Finite State Machine (EFSM) and Specification Description Language (SDL). A set of tools [Alur R 1994], [Richard 1998], [Tahat 2001], [Rayadurgam 2001] have been developed based on these techniques to generate system level test suites from system models. [D.Deng et al. 2004] have developed a semantic model using which the software testing and maintenance can be carried using the inputs obtained from all the stages of the life cycle of the software development.
Test tube [Chen 1994] techniques are based on the coverage dependencies between the test suite and the program. Dependency is established between the test cases and the programs. Any time a change is made on the programs, the affected identities and their associated test suits are identified. [Tsai 2001] have proposed scenarios based functional regression testing based on the END-TO-END integration testing scenarios.

Most of the techniques are suitable for testing the loaded systems and the aforesaid techniques have not considered the event based dynamic nature of the embedded systems for online testing and software maintenance. The techniques proposed are offline and do not completely support the online nature of the embedded systems. These models available in the literature have not even considered the need for undertaking the testing at different locations such as HOST, Target, and both Hardware and Software and using different methods of testing.

[K Rajasekhar et al 2007-6] have proposed a Semantic Architectural model that help identifying the framework required for undertaking the comprehensive testing of the embedded system.

### 3.3.1 Semantic Model for Embedded Systems

In this section a semantic model for software development and testing is proposed based on the object/Procedural oriented analysis and design modeling language and other languages using which the data repositories and the process models are built. This model covers all the stages of software development, testing and software maintenance. The semantic model is presented in Fig 3.3.

The semantic model comprises of three computational blocks. The first block is related to the models that best represent the analysis and design of the system. The second block is related to all the models which are required
for test case generation and the third block is related to the models required for undertaking the testing using test cases generated through models described in the 2nd block.

The requirement of the embedded systems can be specified in terms of use case diagrams, and use case descriptions. The class diagrams represented the business objects which encapsulate the attributes and the functions. The use cases are realized through the interaction between the business objects. In the case of procedural models, functions encapsulate the attributes and the interaction between the functions is achieved through function call sequences which are used in the chapter 4.0 to develop the pilot project. The state charts provided the state transitions that an embedded system will undergo due to the occurrence of the events externally.

Fig 3.3 Semantic model for Software Development, Testing and Maintenance
The functional modules support processing an external input or activation of an external output. The tasks are the recognizable units of processing under a real time operating systems. The tasks provides for processing within itself and also through a chain of function call sequences. The statement files contain the code and are related to the functions.

Thin threads specify END-TO-END processing when events occur and the threads are tested through patterns which provide input and output conditions. The patterns specify the pre and post conditions.

The software implementations can be represented in terms of a database connecting the classes, its methods, attributes and the statements contained in the methods and relate the objects to represent thin threads.

The comprehensive testing of embedded systems involves the testing to be carried at different computing locations using different methods by using different process models.

Various test scenarios can be represented into a template model as suggested by Tsai [Tsai 2001]. From the simple template model complex scenarios can be derived and the same can be related to thin threads which provide the minimal of processing and thereby testing that should be conducted. Thin threads represent the processing when events occurs and the patterns provides for testing the thin threads by specifying the input and output conditions. The Template model describes the sequence in which the test scenarios are to be tested and also specified parent-child relationships.

The test scenarios, thin threads, test patterns provides for testing framework. All the Models stated in the 1st Block will help generating, capturing and maintaining the test scenarios, and thin threads. The models in the 1st block are helpful in constructing the data repositories required for
testing the embedded systems in terms of END-TO-END testing and regression testing. The process architectural models provide the frameworks which should be used for undertaking the actual testing of the embedded systems.

For comprehensively testing the embedded systems, various test case types that should be used are to be identified and the same are represented through various test case types which can be used to undertake testing by using one or more methods (Scaffolding, Simulators, Assert Macros, third party tools, emulators and monitors etc.).

The test generators will be able to generate the test cases based on the Event, Location, Test Method and type of testing to be carried which may be functional, Integration, END-TO-END or regression testing. These test cases can be used by the test processes resident in the target embedded system or the HOST to carry testing.

The third block has all the models required for undertaking the actual testing. The host and the target are provided with the processes that help in undertaking the testing. The processes that are used at the target include in-circuit emulators and monitors. The HOST is provided with the processes such as Scaffold Code generators, Macro code generators, script generators for third party tools and instruction set simulators. The third block also provides the processes required for undertaking the test analysis and test completeness.

The host based system should also communicate with the embedded system to configure the environment parameters, the boundaries within which the production system should operate.

The semantic model presented in Fig 3.3 provides the complete framework for undertaking the comprehensive testing of the embedded systems.
All the models discussed in semantic model are discussed in this chapter in detail.

### 3.3.2 System Architecture

System architecture model for testing of an embedded system is shown in Fig 3.4. The Application development related details which include use case diagrams, class diagrams, Class files, Attribute files, function files, application tasks, function executions sequences, and the statements are captured and maintained in central Data repositories as depicted in Fig 3.5 below.

![Semantic Model for Testing Embedded Systems Related to Mission Critical Systems](image)

**Fig 3.4 Semantic Model for Testing Embedded Systems Related to Mission Critical Systems**

The relationships between the classes and the sequence in which the classes are executed to realize the use cases and the relationships between
the classes, tasks, functions, and statements are shown in the Fig 3.6. In the event if development is done using the procedural model then the mapping of the use cases to the functions is not required. The sequence in which functions are executed is modeled through functional call sequences. Application tasks are also the execution units identified under any of the real time operating systems and the functions are related to the Application tasks and the functions are executed in a particular sequence to realize the tasks. Fig 3.6 shows the relationships.

Fig 3.5 Code related Data Repository

The testing requirements are presented as a set of test case scenarios and complex scenarios are constructed from elementary scenarios. A complex scenario is related to thin thread. Thin threads are processed by way of interaction among various tasks. Thin threads are tested through patterns and the patterns provide Pre and post conditions required for testing. The patterns are associated with events that actually invoke one or more of the thin threads.
Thin threads are tested at different locations and using a specific testing method. The relationships described are shown in the Fig 3.7. Different types of test cases that are to be used are modeled into an entity and the relationship between the test case types, Thin Threads and Test methods are stored in the central repository. The relationships are presented in Fig 3.8.

The Data repository derived from the semantic models help provides the framework for generation of test cases and undertaking the testing of the embedded systems. The Embedded application can be recognized as a set of relationships and the relationships can be modeled into a set of entities.

- A class is a structure
- A Class has defined attributes
- A Class has Functions
- A Functions has arguments and may or may not return a Value
- A Function can call one or more Functions
- A set of statements are contained in the functions and the Tasks
- A use case is realized through a sequence Behavior
- A sequence of execution is achieved through interaction between the functions of a set of classes by passing message between them
- A set of Classes are grouped into Tasks
- A set of Functions are related to tasks
- Functions and tasks have a set of Statements
- Attributes are referred in Classes, Functions, Tasks,
- A set of tasks are related to a Thin Thread
- A Thin thread is processed and tested through a Pattern
- Patterns define the occurrence of events
- Applications are developed using a set of tasks and functions

The Test environment is also modeled through modeling a set of entities and establishing a set of relationships among them.

- A thin thread is tested via a set of test methods
- Test methods are used at different testing locations
- Test requirements are projected through a set of scenarios
- Elementary scenarios are grouped into Complex scenarios
- Complex scenarios reflect the thin threads
- A test case type is related to a Thin thread and method of testing

Fig 3.7 Thin Thread Based Data Repository
• Interface to third party tools is established through interfacing commands

![Fig 3.8 Test Case based Data Repository](image)

A set of relationships can also be established among the data elements and the statements.

• A data element is associated with another data elements either by expression or assignment
• A statement has relation with a set of data elements
• Statements are related to either tasks or Functions

The relationships between the data elements and the statements will find the traceability of all the data elements that have been affected. The traceability of statements to the functions and to tasks and then to the thin threads can be achieved. The effected thin threads can then be chosen for undertaking the regression testing.

The scenarios based Test development process models capture information related to test scenarios and their inter relationships. Complex scenarios are developed from basic scenarios and the Basic scenarios are
tested through testing of thin threads. The thin threads are tested through a set of test case types which are used for testing through various test methods.

Patterns provide the Pre input and Post output conditions which are used to generate test cases that should be used for testing the Thin Threads. Patterns are related to events that must be processed through the thin threads. A thin thread is the minimal execution that must be undertaken when an event occurs.

The test generators generate the test cases that should be used at different locations and the test methods that should be used to undertake the testing and the test cases generated also are stored in the central data model.

HOST based model will have the code that undertakes the testing within itself and also testing the hardware through the Hardware testing Models. HOST based system also carries testing of the Type HOST+TARGET. The test execution processes configures the Application execution environment through test processes which sets the environment by interfacing with the Real time operating system (RTOS).

3.3.3 Test Case Generation Process

Using the data model presented at the section above, a Test case generator can generate the test cases by looking at the Inputs and outputs associated with Patterns and looking at various testing techniques such as partition testing, boundary value testing, and equivalent partition tests. Each Test case is associated with several test data. Several techniques can be applied to different data elements and generate the test cases.

Test case generator can also generate the test cases which can be identified and classified based on the location at which the testing should be
carried and the method with which the testing be carried. Again testing is carried by way of following a pattern. A test scenario is tested by testing a set of test cases. A test case is tested at a location by using one of the methods which include testing through Scaffolding, Instruction set simulators, or through the Assert Macros, or through Logic Analyzers or Incircuit Emulators or software only monitors. A particular type of testing at any of the locations through any of the methods may be carried by following a Test pattern. Patterns of testing involve usage of a particular test process.

3.4 Architectural Frameworks of Process Models

Several architectural frameworks presented in this section are used for test case generation and testing of the embedded system.

3.4.1 Architecture for comprehensive Testing of embedded systems

Comprehensive testing of embedded systems requires an architecture that includes test methods, test equipment and processes that can be used to carry testing of an embedded system. The Architecture required for conducting the comprehensive testing is presented in Fig 3.9.

User interface in the HOST will not only interact with the End user but will be able to interact with all the testing methods proposed in the HOST. The user interface at the HOST will interface with the Communication modules at the Target and the Logical Analyzer for initiating the required testing and obtaining the test results and storing the same at the HOST for conducting the test analysis subsequently. The way a test case is used to do testing is dependent on whether testing is done at HOST or Target or at both at Target and HOST.

A test scenario is tested by testing a set of test cases. A test case is tested at a location by using one of the methods which include testing
through Scaffolding, Instruction set simulators, or through the Assert Macros, or through Logic Analyzers or In-circuit Emulators or software only monitors.

3.4.2 Testing at Host

The ES software can be tested at the HOST at which the ES code is developed. Testing at HOST primarily tests the Hardware independent code. The Hardware dependent code is to be scaffolded to test the Hardware Independent code. The test methods used at the HOST include Testing through Instruction set simulators, testing through assert Macros and testing through Scaffolded software. Each of the methods will help testing different kinds of testing requirements.
The following are the sample test cases that are to be tested using the testing methods through Instruction set simulators, Assert Macros and Scaffolded Software.

- **Scaffolded Software Test Cases**
  - Testing Interaction with peripheral devices bypassing interrupt service routines
  - Testing Time based Functions by calling the interrupt service routines directly
- **Assert Macro Test Cases**
  - Checking for proper usage of data bits
  - Checking for Proper function calling sequences
- **Instruction Set Simulator Test Case**
  - Testing the portability
  - Testing for Response time

The process model presented in fig 3.10 captures the test Scenarios and the test cases are generated. The test cases are segregated based on the location at which the testing should be carried and the test method using which the testing must be carried. The test cases are used to modify the original test code to either scaffold the code or use Assert Macro within the code. The changed code is then compiled and executed to generate the test results. In the case of testing with instruction set simulator the test data is generated and the ES code is tested under the influence of instruction set simulator using the test data generated for that purpose. All the test results are accumulated and test coverage analysis is carried.

### 3.4.3 Testing the Target Hardware

The target is developed by designing and developing the hardware and then moving the software into the Hardware leading to development of an embedded System.
Hardware based testing cannot be tested using the software. Separately identified test equipment that runs software such as logic analyzers are required to carry testing of the Hardware of the target system. Logic Analyzers help testing the complex mechanisms at signal level. The timing of the signals, the validity of the signals over certain time periods, the sequence in which the signals are generated and processed, must be tested. The issues of triggering of the signals whether level triggered or edge triggered must also be considered. Using Logic analyzers proper functioning of all the Hardware components can be tested and the test results can be stored within the Logic Analyzers.

Fig 3.10 Process Model for Testing at the HOST
Various parts of hardware are connected to the Logic Analyzers through Probes. The probes can be connected to a Logic Board which supports several channels. The Logic Board can be driven by command issued from a Personal Computer. The Logic Board can be connected to a personal computer (PC). The software resident on the personal computer can issue commands and data that represent the test case to the Logic Board and the logic board after carrying the testing will make available the test results back to the PC. The Block diagram illustrating the connectivity Testing gadgets to the embedded system and the HOST based system is shown in Fig 3.11

Fig 3.11 Hardware Testing through Logic Analyzers with HOST Interface

The following types of testing can be carried using the Topology described above

- Timing of signals
• Sequence of Occurrence of the signals
• Pattern of occurrence of signals
• Validity of the signals
• Occurrence of the uncommon events
• Time during which the devices are accessed
• Test the mapping of the signals associated with a cross section of the code.

The process model that relates to testing the hardware is shown in Fig 3.12. In this model test cases are generated from test scenarios and the test cases that relate the testing of hardware is segregated based on the patterns of testing required. Test data is then generated using the test case in terms of the Logic Analyzer Data and the Commands. The HOST system then communicates data and commands to Logic Analyzer’s HOST. The testing is carried by the Logic Analyzers’ HOST as per the commands and test data provided and the test results are communicated back to the HOST.

3.4.4 Testing the Target with HOST

Some part of the testing which includes testing the inbuilt devices within the micro controller, testing for memory leakages, testing for function by function usage analysis, testing for weak code, testing for highly used functions etc. cannot be done either at HOST or at Target. These Kinds of Testing requires a process model that involves both HOST and Target. The target in this case will have In-Circuit emulator which will have the code to work in conjunction with Target Code. In-circuit emulators replace the Micro Controllers in the target machine.

The emulation software is placed in a separate chip. The emulation software provides the necessary interface required to test the ES code. The Emulator provides for parallel processing and runs even in the event of failure of the main application. The emulator provides for testing and
debugging facility. The In-circuit emulator can be used when it is included in the overall strategy of testing and debugging.

The process model that includes the in-circuit emulator is shown in Fig 3.13. The test cases that should be executed using the HOST-TARGET location using the In-Circuit emulation mode are first segregated on the HOST. The test cases are used to generate test data and the patterns that should be used to carry testing on the target with data and test pattern sent from the host through communication software.

The communication component on the target will provide the interface required to interact with ES application code and the In-circuit emulator to initiate execution of the test and collecting the test results. The test results are sent to the HOST where coverage analysis is carried.
The above process models help in understanding the process involved in undertaking the testing from the point of view of test case generation, Test case submission, test execution, test reporting and conducting test analysis.

3.5 Architectural Framework for Building Test Scenarios

Use Case oriented software engineering has gained wide attention since early 1990 [Jacobson 1994]. Use cases help communication among customers, software analysts, designers and developers. Use cases represent the system usages without the necessity of referencing the design of the system.

Use cases can be instantiated into several scenarios. A Scenario represents a partial execution of a system and can be represented as a set of message flows within the system.

The literature survey reveals the existence of several techniques that help defining and using the use cases and scenarios. The techniques include UML (Unified modeling language) [Booch 1991], UCM (use case Maps) [Amyot, 2003], MSC (message sequence charts) [Lee NH 2003]. The UCM and MSC are quite often used for identifying the usage scenarios related to Real time systems. While MSC is closely related to a state model of an embedded system, UCM can be used to map a system structure and do analysis using the structure.

Use cases are high level descriptions of the systems. Jacobson recommended that number of Use cases should be in between 70-90 in the case of larger systems and in between 20-50 for normal system. In contrast to use cases scenarios are detailed descriptions with detailed design information encapsulated into them. There may be thousands of scenarios for an integrated system. A single test case can be related to several of the test case scenarios.
Software requirements specifications provide the foundation for Analysis, design, development, testing and deployment. It will be largely helpful to find a model to present the requirements in a formal way [Bai 2002]. Scenarios can be derived from system requirements and they often represent the functional requirements from the end user point of view.

The scenarios can be classified as atomic or complex. Atomic scenarios help in verifying the individual functions by conducting the unit testing and complex scenarios are helpful for conducting the integration testing, END-TO-END testing, regressions testing [Paul 2001-1, 2001-2] and stress testing. Complex scenarios are helpful to test a set of multiple functions of the system.
3.5.1 Identification and modeling test scenarios

Identification of test scenarios from the Requirements specification is the key issue. [Tsai 2001] recommended a hierarchical structure of representing the scenarios. The scenarios are hierarchically organized for better understanding and deriving of the actual scenarios. The hierarchical organization reflects the functional dependencies between the scenarios and the composition/decomposition relationships between the scenarios. The following basis can be used to realize the hierarchical relationships.

- A basic function is represented by an atomic scenario
- Related scenarios can be grouped into a scenario group level by level
- A scenario is decomposed into basic scenario level by level
- Scenarios are connected through control constructs
- Compose scenarios into complex scenarios

Temperature Monitoring and control system

Functional Requirements

Testing for starting the pump when the Actual temperature is more than Reference Temperature

Testing at the Host

Testing Through Instruction Set Simulator

Testing for the functioning of the pump through simulators (unit Testing)

Testing for the functioning of the Temperature sensor through simulators (Unit Testing)

Testing for Proper reading of the Reference Temperature from the Host through simulator (Unit Testing)

Testing the starting of the pump when the Actual Temperature is more than the reference temperature through simulators (Integration Testing)

Testing for the response time of Starting the Pump (Integration Testing)

Testing for authorization to access the system (Integration Testing)

Testing at the Target through Logic Analyzer

Testing at the Target along with Host

In Circuit Emulator

Through Monitors

Fig 3.14 Hierarchical Presentation of Test Scenarios
Scenarios can be grouped based on the location where the testing should be carried or based on the type of testing to be carried. For testing the scenario S5, the scenarios S1-S4 may be grouped with S5.

Another way of identifying the scenarios is by doing the functional decomposition of the system by dividing the system into distinct modules and the modules into functions and further down. The overall functions of the system can be divided into distinct features at module level wherein each module is represented by high level scenarios. The high level scenarios are further divided level by level until the basic scenarios are identified.

Fig 3.15 shows the decomposition for a Temperature Monitoring and Control System.

![Functional decomposition of the Temperature Monitoring and control System](image)

**Fig 3.15 Functional decomposition of the Temperature Monitoring and control System**
The system is decomposed into three levels in hierarchical manner and the decomposition is presented in the Fig 3.16. The group scenarios are decomposed into basic scenarios and the Basic scenarios are decomposed into subgroup scenarios.

Scenarios may share some functionality that is commonly used across the system such as system authentication. Group scenarios or the modular representation of the system generally requires interaction between them.

The modular interaction of the scenarios will provide the integration testing requirements of the system.

Sub scenarios decompose a scenario into relatively independent functional parts which are atomic functions which can be executed in isolation without having any dependency with any other function. Table 3.1 shows scenario-Sub scenario identification for hierarchical diagram presented in Fig 3.16.

![Fig 3.16 Scenarios decomposition of the Temperature monitoring and control system](image)
The sub scenarios are identified in such a way that they are identified by the location of testing and the method with which the testing is carried. Sometimes it is possible that the same scenario be executed at different locations and testing of the scenarios is done using different test methods.

### 3.5.2 Risk Assignments to test Scenarios

<table>
<thead>
<tr>
<th>Group Serial</th>
<th>Group Scenario</th>
<th>Basic Scenarios</th>
<th>Sub Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pump Control System</td>
<td>Start Pump-1 when Reference Temp-1 is less than the reference Temp-1&lt;br&gt;Stop Pump-1 when the Reference Temperature-1 is more than the actual Temperature 1&lt;br&gt;Start Pump-2 when Reference Temp-2 is less than the reference Temp-2&lt;br&gt;Stop Pump-2 when the Reference Temperature-2 is more than the actual Temperature 2</td>
<td></td>
</tr>
</tbody>
</table>
| 2            | Communication System | Communication with Host  
Send Temp-1 to host  
Send Temp-2 to Host  
Receive Temp-1 from HOST  
Receive Temp-2 from HOST  
Communication between A/D and the Controller  
Read Temp-1 from A/D through I2C communication  
Read Temp-2 from A/D through I2C communication  
Communication with LCD  
Read  
Write  
Initialize |                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 3            | Temperature Processing System | Compare Temp-1 and Temp-2 and trigger the Buzzer  
Compare Temp-1 with the reference and Start or stop the pump-1  
Compare Temp-2 with Reference and start or stop the Pump-2  
Read Temp-1  
Read Temp-2 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
There is a certain element of complexity and Risk with which scenarios based testing is done based on the choice of location and testing method used. The complexity of testing of the scenarios can be generally be grouped into Low, medium and high. Similarly the Risk of undertaking the scenarios testing through test cases is dependent on the choice of location for testing and the choice of method for undertaking the execution.

The Table 3.2 shows the complexity and Risk of undertaking the testing of the Scenario at different locations and test execution method. The testing strategy to be adopted for testing the scenario will be the location and the method which involves the Complexity and the Risk as low.

Various alternatives of testing the scenarios may be graded using the weighting factor derived by multiplying the Complexity with the Risk. The weights to be considered can be as High (3), Average (2), and Low (1). The lower the weighting factor the better will be the choice of selection of the Location and the method of testing the Scenarios.

The risk however is the combined risk of execution of a set of software components and the Hardware associated with the execution of the Software. A scientific analysis is to be carried to assess the combined Risk by conducting the reliability analysis of Hardware and software which is beyond the scope of this thesis.

From the table 3.2 one can see that the scenario at serial number 1 can be efficiently tested by using the logic analysis method at the target. The scenarios at serial number 2 can be efficiently tested using the Target- Host Location by using the Technique of in-circuit emulation.

Risk factors can be assigned to the scenario based on the components involved in testing the scenarios and the failure rates of those components. The impact assessment of any failure occurring is equally important. An
embedded system as such should not fail. The embedded system when fails, investigating the cause for the failure and then correcting the failure is time taking. Prevention is better than cure.

It is important that risky logical paths are identified and the scenarios which deals with risky logical paths must be tested thoroughly as the failures will make the embedded system halt fully or partially.

3.5.3 Scenario relationships

The scenarios may be related to each other with respect to their execution paths [Paul 2001-1] [Tsai 2001]. The following are different kinds of relationships that exist in between the Scenarios

<table>
<thead>
<tr>
<th>S. no</th>
<th>Basic / sub scenario</th>
<th>Location of testing</th>
<th>Method used for testing</th>
<th>Complexity</th>
<th>Risk</th>
<th>Weighting factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start the Pump-1 when Temp-1 is more than the Reference Temperature</td>
<td>HOST</td>
<td>Scaffolding</td>
<td>High(3)</td>
<td>High(3)</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assert Macro</td>
<td>Average(2)</td>
<td>High(3)</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target</td>
<td>Logic Analysis</td>
<td>Low(1)</td>
<td>Low(1)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Read Temp-1 through A/D Converter</td>
<td>HOST</td>
<td>Scaffolding</td>
<td>High(3)</td>
<td>High(3)</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target</td>
<td>Logic Analysis</td>
<td>Low(1)</td>
<td>Average(2)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target-Host</td>
<td>In Circuit Emulation</td>
<td>Low(1)</td>
<td>Low(1)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2 Risk Factor associated with testing method and Location of testing

- The execution path of scenarios can be a part of a path of execution of another scenarios (Path Contained)
- Two scenarios have identical paths. In this case they may share a certain set of attributes and conditions (Identical Paths).
• Two scenarios may deal with different paths of execution all together and not related to any other in any way (Independent paths)

Scenarios may also be related based on certain conditions that define the existence of the scenarios. In simple terms, conditions are associated with scenarios. Relationships between the scenarios can also be developed by relating the conditions which are associated with the scenarios.

The following relationships between the scenarios can be derived through the analysis of conditions.

• Two scenarios having conditions which are not related in any way are independent conditions or scenarios
• Two scenarios are related if a condition of scenarios triggers a condition of another scenario
• Two scenarios are mutually exclusive if the condition of one scenario cannot exist when the condition of other scenario exists (Mutually Exclusive)
• A scenario may have more than one condition when they are mutually exclusive

The understanding of the relationships is important to determine the scenario execution dependency and condition dependency. These relationships are important to recognize the scenarios that must be considered for conducting the Integration and Regression testing and also the sequence in which unit testing of the atomic scenario is carried.

Dependency analysis of scenarios is also very important for dealing with the change management which involves conducting of regression testing. Scenarios are dependent on each other in the following ways.
Scenarios are functionally dependent if they belong to the same scenarios group in which the scenarios represent the common functions from different aspects or the scenarios contain same sub scenarios.

Scenarios are data dependent with respect to their template definition. Two scenarios are input dependent if they are triggered by the same data, events and actions. Two scenarios are also output dependent if they produce the same results.

Two scenarios are execution dependant if they cover common software components which are code modules, functions or interfaces.

Two scenarios are condition dependant if they are affected by the same set of conditions.

Two scenarios are reference dependant if they share the same artifacts such as documents.

The scenarios identified will have to be verified for the completeness and Consistency checking.

3.5.4 Building Complex Scenarios

The scenarios and their relationships can be used to eliminate infeasible or useless combinations of complex scenarios. Using the relationships the complex scenarios containing two mutually exclusive scenarios can be identified and eliminated. In fact, the complex scenarios should be generated with sub scenarios that are related or independent.

Complex scenarios are to be modeled to model complex usages of the system [Tsai 2001] or model the order of execution of the sub scenarios that belong to a scenario.

Complex scenarios can be connected using the control constructs such as sequencing, looping, and concurrency and attached conditions. For example,
in the temperature monitoring and control system the following Complex scenarios can be constructed.

If Temp-1 Sensed

If Temp-2 Sensed

If (diff = Temp-1 – Temp-2) > 2 then assert signal to start the pump

In this example the three atomic scenarios are combined to make one complex scenario. Generally the complex scenarios are the scenarios using which the integration testing is carried as they relate to integration of components within the software, or integration of hardware components or integration of software components with hardware components. Complex scenarios are also required for undertaking the regression testing as regression testing normally crosses several execution paths of the system. Construction of Complex scenarios looking from the angle of both implicit and indirect data connectedness of the system is important as the data variables cover the regression testing requirements.

A complex scenario is related to a thin thread. A thin thread is an END-TO-END processing when events are initiated. The END-TO-END processing is achieved through thin threads which are directly related to complex scenarios. At times a simple scenario may be related to a thin thread which is related to patterns of type "Command and response". Patterns provide the basis for processing and therefore provide the Pre and Post conditions.

3.5.4 Completeness, Consistency and Validity checking of the Scenarios

The consistency and validity of the scenarios must be verified before proceeding further with the identification, of its relationships with the thin threads. [Xiaoying B 2002] have proposed the following C&C rules with which the checking of Completeness, Consistency and validity checking of the complex scenarios can be undertaken.
The following rules can be used to determine the completeness and consistency of the scenarios.

- No Scenario of one Group is same as another similar scenario of another group.
- No Scenario is same as another. Every scenario must differ from other scenarios at least by input, output or execution path.
- No sub scenario will overlap with another sub scenario of the same scenario that means that the sub scenarios will have non-overlapping execution paths.
- For each scenario the expected behavior should be identifiable with normal inputs.
- For each scenario there should be two related scenarios that exhibit the behavior under Abnormal Conditions and Exceptional Condition. For example the Exceptional and Abnormal Conditions related to scenario “Sense Temperature-1” could be as follows.

\[
\text{If Temp-1 not sensed within 100 Micro Seconds} \\
\quad \text{Correlate the Temperature from other neighboring Temperatures} \\
\text{Else} \\
\text{If Temp-1 is not in range} \\
\quad \text{Raise an alarm and Correlate the Temperature from other neighboring Temperatures}
\]

The validity of complex scenarios is verified by using the scenario relationships and dependency analysis. The following rules help in verifying the validity of the complex scenarios.

- When Scenarios having associated conditions are used to develop complex scenarios using the conditional operators such as if-then-else,
switch-case, then there should be a dependency between the conditions associated with the scenarios and the conditions with which the complexity is built.

- Scenarios that are connected through sequencing operators must be control dependent.
- Scenarios that are connected using the concurrency operators must be input dependent with respect to triggering action or an event.

### 3.5.5 Identifying Change Scenarios

Identifying the change scenarios that are likely to change is important for simple reason that when change takes place on the items that related to the change scenario, all the test cases that are related to the scenarios can be used for testing.

The design for change process implemented by [Parana 1985] identifies potential changes and encapsulates them in the design at the beginning. Parana suggested that hardware parts, Hardware dependent software parts, parts that deal with input and output and parts that deal with data structures and their related algorithms are most likely to get changed. If these parts are encapsulated into individual classes, then the change if occurs will be limited to the class concerned and the testing can be limited to only those scenarios that are related to the class concerned.

For the parts that are likely to change one can use a suitable design pattern to encapsulate them. When a change occurs, the impact of the change can be isolated within certain classes only and only those classes that are affected can be tested.

When changes to the scenarios are undertaken, the complex scenarios also will undergo change and the connectivity of the changed scenario to the thin threads must also be addressed.
3.5.6 Scenarios Templates

Every scenario must be defined explicitly and captured into a database which becomes the input for generating the test cases. Considering the discussion above, the following Template design is used to fully describe the Template.

The above description of the scenarios can be modeled into a Schema Definition of a relational database. Each of the scenarios is directly related to the Thin Thread.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario-ID</td>
<td>Global reference number that may be generated by the Testing system</td>
</tr>
<tr>
<td>Name</td>
<td>Scenario Name which implicitly identifies the scenario</td>
</tr>
<tr>
<td>Thin Thread ID</td>
<td>The Thin thread that is related to a Scenario</td>
</tr>
<tr>
<td>Parent_Scenario_ID</td>
<td>The ID of the scenario which is the parent of this scenario</td>
</tr>
<tr>
<td>Sub Scenarios</td>
<td>The sub scenarios which are subordinate to the scenarios</td>
</tr>
</tbody>
</table>

Table 3.3 Scenario Template

3.5.7 Scenario Modeling through Data

The ER diagram presented in Fig 3.17 shows the Decomposition of the Scenarios into respective entities through which the relationship between the scenarios is established. A scenario is connected to other scenarios in terms of a Group, subordination, precedence and succession.

Every scenario is related to a thin thread. A thin thread is an end to end processing that should be undertaken when one or more of the events takes place in a sequence. Thin threads are processed according to a pattern which specifies the Pre and post conditions.
When ever the changes are made by adding the data elements, the same are to be populated into the data dictionary. When modifications are made to data elements the same will be affected on the data dictionary. When data elements are deleted, the data elements entries in the dictionary are deleted.

The Changes made to data dictionary can be recorded separately and using this data the threads/scenarios that are associated with the data can be selected and used for the test generation for conducting the regression testing. New scenarios may be generated and stored which will deal with the new data elements added into the system.

The description of the scenarios through a thin thread can be translated into a data model which helps capturing the Test Scenarios, develop the relationships between the scenarios and the thin threads, and also relating the thin threads to code units, events and the sequence in which the code units must be executed when an event is initiated. The ER diagram shows the decomposition of the Scenarios into respective entities through which the relationship between the scenarios is established. A scenario is connected to other scenarios in terms of a Group, subordination, precedence and succession.

A scenario is also related to Code units indirectly through the thin threads of the application and the code units that are related to an application are to be executed in a sequence. The execution sequence is initiated through an event which may be an external event, or an event caused due to passage of time. The Test Scenario is associated with some input elements and output elements. The data repository can be built from the source code. The data repository establishes relationship between the code and the data elements. It also recognizes whether the data is Global or Local. When ever the changes are made by adding the data elements, the same are to be populated into the data dictionary.
When modifications are made to data elements the same will be affected on the data dictionary. When data elements are deleted, the entries of data elements from the dictionary are deleted. The Changes made to data dictionary can be recorded separately and using this data the scenarios that are associated with the data can be selected and used for the test case generation for conducting the regression testing. New scenarios may be generated and stored which will deal with the new data elements added into the system.

The data model will help conducting the ripple effect analysis to identify all the affected areas of the code and the data that is related to the code. The scenarios that are associated with the affected Data elements either directly or indirectly can be identified or used for generating the test cases needed for conducting regression testing.

The data model described above can be represented as persistent classes and the relationships between the classes will be the relations that have been used to relate the entities in the data model.

The data model integrates the data view with the functional view by associating the data with the scenarios/thin threads and also accommodates the change management system within the data model so that it becomes easy to create and identify the scenarios that are to be used for conducting the regression testing.

3.5.8 Generation of Test Cases

A scenario is related to a thin thread which is basically a process that accepts input, processes and transforms data. The data may be generated internally or sensed by various types of the sensors in the system or retrieved from the persistent storage. The data can be even inputted by the user as the test case generation process is essentially done on host where
the user interface can effectively be used. The test results are made available as outputs from the system.

Each scenario and sub-scenarios can be used to test a sub system or cluster of subsystems [Tsai 2001]. Complex test scenarios can be generated from complex scenarios by combining scenarios and sub scenarios. Complex scenarios can be generated keeping in view of the integration testing requirements.

Numerous complex Test Scenarios can be derived based on various criteria which include path coverage, condition coverage and loop coverage [Tsai 2001].
3.6 Architectural Framework for END-TO-END Integration Testing

Embedded systems are event driven. The events are either system driven or initiated by an external environment through sensing devices such as Temperature sensors. The sensed data is transmitted and captured by an embedded system. The inputs are processed and the processed results are used to trigger signals that control the physical environment. The entire processing can be represented in terms of a physical thread of execution. An event process is a functional requirement, and the devices involved need that a function be executed as per certain timing considerations and the timing is to be achieved by following a particular pattern.

Integration testing of embedded systems has been a challenge as several issues of integration have to be considered from the point of view of integration of Hardware, Software and both. Integration of testing of an embedded system may typically be carried either completely on hardware, or software or both.

END-TO-END Testing of an embedded system is more relevant as the entire processing can be recognized in terms of event based processing starting from initiation of an interrupt by the hardware to the processing of emergent requirements and passing the control to mundane tasks which can be executed in a predefined process flow leading to control of physical parameters. This means that the END-TO-END testing can be carried considering the entire system as a whole and not as part of a system.

Given a scenario of an embedded system, it is necessary that the test cases are generated and testing is completed rather quickly, thus reducing the time required for testing the embedded system leading to heavy cost reduction.
Raymond Paul [Paul 2001] presented a model to carry testing using the concept of thin threads which are true representatives of the functional requirements of a system from the end user point of view. This model typically suits to loaded systems. Raymond proposed the generation of test scenarios and test cases using thin threads which are represented as a tree.

Unlike the loaded systems the embedded systems are event driven and the events either occur randomly or in a cooperated sequence. The occurrence of some of the events is conditional. The occurrence of the events can also happen in terms of some patterns. Feng Zhu and others [Zhu F 2002] have presented event patterns based on which the test cases can be generated.

About 80% of time is to be spent on integration testing of the embedded systems as the integration of Hardware Devices, Integration of Software Components and the Integration of software components with the Hardware devices is the major effort to complete the entire testing. The integration testing of the embedded systems must be conducted considering the entire system as a whole. The techniques available in the literature which include Top down and Bottom Up methods [Pressman 2000] [Hans Van Vliet 2000] explore the program or design structure of the system [Kirani 1994-1][Kung 1999] [Tsai 1999] [Tsai 2000].

These methods while they are suitable for embedded systems are not quite applicable for testing embedded systems. [K Rajasekhara et al 2007-2] have presented a Model for undertaking the END-TO-END Integration testing of the embedded systems.

### 3.6.1 Event Processing and Thin Threads

Occurrence of a particular event and the connected processing of the event can be regarded as one of the functional requirement from the end user point of view. All the events that need to be supported as individual
functions can be formed into a tree with each branch regarded as event based processing and the leaves of the branch can be regarded as thin threads representing the related functionality.

A thin thread is a complete trace of data/messages using a minimally representative sample of external input data transformed through an interconnected set of systems to produce a minimally representative sample of external output data. The execution of a thin thread demonstrates a method to perform a specific function. A thin thread is a minimum usage scenario of the integrated system. A thin thread is a complete scenario from the point of view of the occurrence of a specific Event.

A Logical event occurs when a device senses some external event and acquires data related to external stimuli. The device sends across the sensed data as analog signals which are converted to Digital signals by way of using A/D converters and the A/D converters can then send the data to Micro Controller. The A/D converter interrupts the Micro Controller and sends the acquired data through an interrupt service routine that gets executed every time a device interrupts the Micro Controller. The interrupt service routine reads the data from the device and places it in the embedded application memory. The task that waits for the arrival of the external data processes the data and produces the output which is then used for transmission of the signals to the actuators which actually control the external environment. The entire sequence of the END-TO-END processing of a Temperature monitoring and Control system is shown in the figure 3.18

In an embedded system there can be several such END-TO-END Processing requirements. The END-TO-END processing involves the interaction between the Hardware Elements, between the Hardware and Software Elements and in between the Software Elements. To ensure that the process is built properly, test cases should be generated and used to test the process and the test results are to be captured. Test data in this case
could be the input temperature data and the output data could be the data used by the control tasks.

Fig 3.18 END-TO-END Processing of a Temperature Monitoring System

There could be several thin threads in the system and tracing each of thin thread will help in determining the minimal testing that should be carried to ensure the reliability of the embedded system. Thin threads that share some commonalities can be grouped together into thin thread groups. The grouping can be nested. A set of threads at a level can be grouped. A set of grouped threads at a level can be further be grouped at a higher level. All the thin thread groups and the elementary thin threads can be arranged in a tree structure manner, meaning in a hierarchical manner. Fig 3.19 shows the Tree structure for a temperature Monitoring and Control system.

Thin threads at each of the Level are grouped to depict different END-TO-END Views of the system. Thin threads are similar to use-cases [Ebner 1998] describing END-TO-END Scenarios. However, thin threads help in conducting dependability analysis, risk analysis, Coverage analysis, Traceability Analysis, Completeness and consistency checking and also form a type of Framework using which the test cases can be generated and testing is carried.
The thin threads for an embedded application must be identified as Black Box thin threads and white box thin threads. The Black Box Thin threads are identified first at the higher level of tree structure from the point of external view and the White Box thin threads are identified at the lower levels of the tree which form the internal view of the embedded system. While the Black Box thin threads can be identified based on the correct input values and output values, the white box thin threads can be identified as a series of failure conditions with a proper input value and a series of failure conditions based on inappropriate output values.

The thin threads can be related to each other based on their execution paths. The thin threads may be independent, identical or contained within one another. For example, the successful sensing and processing of the Temperature-1 and Temperature-2 thin threads are identical. The thin

---

**Fig 3.19 END-TO-END View of Temperature Monitoring and control System**

The thin threads for an embedded application must be identified as Black Box thin threads and white box thin threads. The Black Box Thin threads are identified first at the higher level of tree structure from the point of external view and the White Box thin threads are identified at the lower levels of the tree which form the internal view of the embedded system. While the Black Box thin threads can be identified based on the correct input values and output values, the white box thin threads can be identified as a series of failure conditions with a proper input value and a series of failure conditions based on inappropriate output values.

The thin threads can be related to each other based on their execution paths. The thin threads may be independent, identical or contained within one another. For example, the successful sensing and processing of the Temperature-1 and Temperature-2 thin threads are identical. The thin
threads that are related to a failure of a device are contained within the thin threads that deal with unsuccessful sensing and processing of Temperature-1 and Temperature-2. The relationships between the thin threads are useful for scheduling the test execution by using the test cases in a proper order. The thin thread tree explains the paths of execution and one can derive the critical paths from the available paths. A critical path is the one that if it fails, it causes the failure of entire system. The critical paths should be tested thoroughly and as early as possible. The independent paths must be tested to complete the coverage.

3.6.2 Processing Patterns

A well structured system is full of patterns. A pattern provides a common solution to a common problem in a given context [Booch 1991]. A mechanism is a design pattern that applies to a group of classes. Patterns are used to specify mechanisms and frameworks that shape the architecture of the system. A mechanism is a design pattern. A framework is an architectural pattern.

Several types of patterns exist which include responsibility chain pattern, Model View controller pattern, Black Board pattern, template pattern, publish and subscribe, pipes and filters etc. In a well structured system, one will find lots of patterns at various levels of abstractions. Design patterns specify the structure and behavior of a group of classes where as architectural patterns specify the structure and behavior of the entire system.

Patterns help to visualize, specify, construct and document the artifacts of the software intensive systems. Embedded systems as such are becoming more and more software intensive systems. Patterns will help defining the internal mechanisms of embedded systems effectively. Patterns provide help in visualization, specification, construction and documentation of a system. Test patterns as such help us design and conduct testing in a more meaningful manner.
A test pattern is a predefined verification mechanism that can be used to test a group of testing scenarios which cause similar behavior.

Patterns are common solutions for common problems. In embedded systems, the processing sequences can be recognized as a pattern. Every thread can be processed through a given pattern specifically when there is a relationship between the inputs fed and the outputs generated especially in relation to the timing of processing. Timing is the most important characteristic of real-time embedded systems. Most of the requirements of the embedded systems are specified in terms of certain timing requirements. The testing mechanism that should be used is dependent on the timing requirements. The timing requirements of the embedded systems can be stated in several ways.

[Zhu F, 2002] has identified different kinds of timing patterns which include basic pattern, Key-Event Pattern, Timed Key Event Pattern, Key Event Driven with time Sliced Pattern, Command Response Pattern, Mode Switch Pattern, Look Back Pattern, Interleaving Patterns and other types of the patterns. For each of the occurrence of a pattern the testing mechanism can be prefixed. If any of the thin threads has an associated timing pattern then we can say that the thin thread can be tested using a verification process that relates to a timing pattern. A specific type of timing requirement requires that the testing be done using a predefined mechanism. A verification pattern is a predefined verification mechanism that can be used to verify a group of behavioral requirements that describe similar temporal patterns or cause and effect relations. Verification patterns can be defined as a set of processes that can conduct testing of a thin thread. Each of the thin thread can be viewed as a requirement pattern and the requirement pattern has an associated Verification pattern.
Basic verification pattern stipulates that an output is expected before a dead line of time after a precondition is met. For example, LCD must be written within 10 Micro Seconds after the Temperature-1 is sensed. The Key-Event pattern states that after occurrence of an event, several of the outputs should be produced in some timed sequence. For example, after the Temperature-2 is sensed, the sensed temperature is transmitted to the Host and also displayed on the LCD. The timed Key event Pattern deals with occurrence of a set of events in a predetermined fashion at the set time intervals and the occurrence of the patterns will lead to one or more number of outputs. For example, after the Temperature-1 is sensed, Temperature-2 must be sensed and the controlling of the starting of the pump and writing the temperature to LCD must be done within 10 Micro seconds of time. Entire processing must be completed within 100 Micro seconds for the Nuclear reactor System to function properly.

The process flow leading to these requirements involves several thin threads and a group of thin threads formed out of all the elementary thin threads. Fig 3.20 shows formation of group threads based elementary threads.

In the case of Key-Event driven Time sliced pattern, after the key event has taken place, the outputs must be generated at a specific time interval. This kind of a requirement arises in case of some Embedded Systems. Command response pattern is one of the commonly used patterns. Some of the embedded systems have human interface using which commands are issued and the embedded system must respond to the issued command within a specified period of time. The system will also have the interface to cancel the command after it is issued. Look-back pattern is a pattern where the embedded system application itself checks for the existence of an environment within the set period of time before the occurrence of an event.
Mode switch pattern helps switching the embedded system from one form to the other such as switching between the production mode to testing mode and vice versa. This kind of a pattern is useful when Host based command driven system is interfaced with the embedded system to support online testing of the system and as such achieves the HOST based control of the Embedded System. Interleaving patterns are based on the relationship between the Inputs and outputs such as occurrence of an event based on the output generated due to occurrence of another event. Several such patterns exist and the testing is solely dependent on the patterns of occurrence of inputs and outputs.

![Fig 3.20 Group threads that relate to Timed Event Pattern](image)

3.6.3 Identification of Data Related to Thin Thread

Each of the thin threads identified to represent the functional requirements of the embedded systems is associated with the data which is either input or output. In the case of the Temperature monitoring and control application stated above, the inputs include Temperature-1 and Temperature-2.
Temperature-2 and the Outputs include the data written on to LCD and a binary output for starting and stopping the pumps and the voltage asserted on the buzzer. For each of the thin threads both the inputs and outputs are identified by studying them individually and combined.

3.6.3 Identification of Conditions through Patterns

Conditions are predicates that effect the execution of the thin threads. A thin thread is executed if and only if all the input conditions are satisfied.

The execution of the thin threads is dependent on the kind of input data selected for the inputs such as a Temperature sensed is within a range of 23-255 degree Centigrade. Several such conditions that must hold well could be traced by studying the patterns. The conditions that could be traced from a pattern include the following.

- Sequence of occurrence of events by way of identification of priorities assigned to the interrupt service routines
- Timing of occurrence of events which include minimum and maximum allowable delays, sampling rates, the time duration in which the Controlling functions must be completed
- The data ranges allowed for sensed data, the allowable data ranges for output data, the voltage levels for asserting the output signals
- The Communication protocol to be used for communication between the Hardware devices and in-between the Hardware devices and the software.
- Availability of the environmental data such as reference temperatures, temperature gradients etc.

All the conditions that need to be satisfied are identified through a study of patterns as the patterns involve the sequence in which the events are executed and the input preconditions and the expected outputs and the
Timing of inputs and outputs can be identified from the patterns. The data conditions can be identified from the following perspectives.

1. Data sensing perspective includes the study of sensors and identifying the range of values sensed and the ampere ranges/voltage ranges that correspond to the sensed temperatures.

2. Data Sampling Perspective includes the frequency with which the data is sensed and the rate at which the data is transmitted to the Micro Controller.

3. Data Transformation Perspective is related to A/D conversion of converting the Amperages/Voltage into data equivalent before transmission of the same to the Micro Controller.

4. The user Interface Perspective identifies the conditions that apply when the data is to be inputted through user interfaces.

5. The communication perspective is related to identifying the communication conditions from the perspective of physical connections between the Hardware components, between the hardware components and the software and in between various software components.

6. The Data Communication Perspective is related to the data conditions that should exist when the software components communicate with each other.

The conditions identified by using any of the perspective can be related to each other. While some conditions are independent, some are mutually exclusive and some conditions are triggered by some other and some conditions are related to each other. For example, the control task generates the control data in a given range for controlling the temperature provided the input data is within a specific defined range. The conditions will provide the basis for conducting the integration testing through the identification data that gets exchanged between the processes and the hardware devices. The conditions will also help in conducting the regression testing as any change in
the conditions will affect the processes that use the changed conditions. When a condition connects one thin thread to other, then any change in one thin thread will affect the other thin thread. All the conditions can be presented as a tree. While the branch nodes represent the collection of the conditions, the leaf node represents concrete conditions.

The condition Tree for the Temperature Monitoring and control System is shown in Fig 3.21

![Condition Tree for Temperature Monitoring and Control System](image)

**Fig 3.21 Condition Tree for Temperature Monitoring and Control System**

The relationship between the thin thread tree and the condition tree can be established by relating the nodes of both the trees with one another. A set of thin threads can be related through conditions. The conditions impose constraints on inputs and outputs and the sequence in which the events occur. If the condition tree is exactly similar to thin thread tree then the trees are well connected. If any of the nodes are not connected properly
either extra thin threads or extra condition branch nodes or leaf nodes are created.

3.6.4 Test Scenario and Test Case Generation

Each of the thin threads is an atomic execution and presents a test scenario. Test scenarios can be created by way of mixing the Thin Threads forming the complex scenarios. Complex scenarios generally involve the integration testing. Complex test scenarios can be formed either through sequencing, looping or through identification of alternate execution.

Sequencing involves testing one thin thread after another. Looping involves testing a single thin thread several times. Looping of the thin thread is useful when a process is to be tested with data sensed by the sensors at different intervals of the time. The complex scenarios can be tested with alternate execution paths. Complex scenarios can be formed from the atomic and the Group thin threads. The complex scenarios can also be formed by connecting several atomic scenarios using if-then-else logic.

For testing each of the scenarios, test cases can be generated by way of mapping the thin thread data with condition data both at the input level and the output level. As such the intermediary level can also be included in the test design as the data is used for establishing communication between different tasks. Inputs can be classified into different types and are associated with different conditions. Similarly the outputs also can be of different types and then can be classified into different types. Table 3.4 shows the input-output mapping using which the Test cases can be generated and the data is used for testing the embedded system through a separate verifier created for the purpose of testing the system.
### Thin Thread: Test the Occurrence of the Thin thread related to sensing Temperature-1 and controlling the starting and stopping the pump

<table>
<thead>
<tr>
<th>Input</th>
<th>Condition-1</th>
<th>Condition-2</th>
<th>Condition-3</th>
<th>Output</th>
<th>Condition-1</th>
<th>Condition-2</th>
<th>Condition-3</th>
<th>Expected Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp-1</td>
<td>Should be in between 23-255</td>
<td>Must be Triggered every 10 Micro Second</td>
<td>Must Occur before the Temp-2 Event Occurs</td>
<td>Control Data</td>
<td>Should be 0 or 1</td>
<td>Should assert signal within 100 Micro Seconds</td>
<td>Temp-1 Must be more than the reference Temperature</td>
<td>Start the Pump</td>
</tr>
</tbody>
</table>

*Table 3.4 Mapping Input-output data with the condition data for test case generation.*
3.7 Architectural Framework for Regression Testing of Embedded Systems

Regression testing is the re-execution of a subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression testing is any type of software testing which seeks to uncover regression bugs. Regression bugs occur whenever software functionality that previously worked as desired no longer works in the same way that was previously planned. Typically regression bugs occur as an unintended consequence of program changes.

Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged.

Experience has shown that as the software is developed, this kind of re-emergence of faults is quite common. Sometimes the reemergence of faults occurs because a fix gets lost through poor revision control practices (or simple human error in revision control), but just as often a fix for a problem will be “fragile” – i.e. if some other change is made to the program, the fix no longer works. Finally, it has often been the case that when some feature is redesigned, the same mistakes will be made in the redesign that was made in the original implementation of the feature.

Therefore, in most software development situations it is considered as a good practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program. Although this may be done through manual testing procedures using programming techniques, it is often done using automated testing tools. Such a ‘test suite’ contains software tools that allow the testing environment to execute all the regression test cases automatically; some projects even set up automated systems to automatically re-run all regression tests at specified intervals and report any regressions. Common
Regression testing strategies can be of two types. Either software can be retested with all test cases (re-test all) or with a subset of the test cases (selective regression test). Selective regression testing can be used to select enough test cases to reveal all failures, or minimal number of test cases or select test cases that only traverse the modified paths of a program. Retesting software with a subset of test cases can reduce the cost of testing the software, and is therefore the most common approach suggested in the literature. A framework is presented for carrying multilevel regression testing.

One issue is that although the re-test all strategy is costly and time consuming, it is not always desirable to find a subset of test cases, especially for those companies that must use retest-all method because of certain constraints such as safety-critical programs etc. Examples of other issues can be due to the use of tools for automation when regression testing is used extensively and frequently. A drawback of regression testing is that the size of the suite of test cases increases when the software is maintained and this makes testing even more time consuming.

There are two types of regression testing which are corrective and progressive. Progressive regression testing is caused by modification of both code and specification, where as corrective regression testing only comprises of code modification.

When using regression testing of type selection technique, the basic concept is to test only the modified parts of the program, but this can lead to undisclosed failures since not all test cases that possibly reveal failures are re-executed. There was extensive research on regression testing techniques and most of them address the regression selection problem. Many of the
algorithms aim to select test cases where the new and the old version of the program differ in output. Others are concentrated to achieve certain degree of coverage. A technique is proposed that uses both of these approaches. The proposed approach is based on two techniques: minimization and test case prioritization.

Most of the existing regression test techniques are using structural based test cases (white box testing). This is because most of them compare the structure of the old program with the new program, and then the tester tries to get the same coverage percentage as the previous test.

Most regression testing techniques concentrate on the test selection problem, ignoring other important issues of regression testing. Equally important to the selection problem is the issue of what triggers the regression event, should tests be executed periodically or at some predetermined instance, for example, after all changes, or after modification of critical components, or during final testing. The hypothesis of the work is based on the fact that the amount of changes between regression testing sessions affects the cost effectiveness, since the test suit grows with each modification leading to selection of more test cases in every session. The impact of this is that the effort to select failure revealing test cases also increases and makes selection algorithms less cost effective.

There exist numerous regression test selection techniques, which include retest-all techniques, random/ad-hoc techniques, minimization techniques, safe techniques, data-flow/coverage techniques, and prioritization techniques.

In the retest-all method, as the name indicates, all the previous test cases are used in the regression test phase. This can, however, be acceptable if the program is small and the number of re-tests is small. For larger software projects and where regression testing is used frequently,
rerun of all test cases is not acceptable. If we have to consider the cost of running test cases or the amount of test cases is too large we can use a regression test selection techniques that select a subset of the previous set of test cases.

By randomly selecting a subset of test cases there is no guarantee that those test cases that need to be rerun are in the subset. Testers with a priori knowledge of the system can see to that if some test cases are missing, these test cases can be added to the test suit.

In minimization techniques the goal is to select a minimal subset of test cases, where each test case corresponds to the impact of the modification in the program. The method selects at least one test case that executes every modified or added statement.

Safe techniques select test cases necessary to reveal all faults in the modified program. This can lead to that in some cases, the retest-all and safe techniques select the same set of test cases. It is proposed that on an average the safe method selects 68% of the test cases. If the safe method selects all test cases, then it is less effective than the retest-all technique since in safe method an analysis is done to determine what modifications have been made. By using regression testing techniques that select potentially revealing test no priori knowledge is needed as in techniques that concentrate less on coverage criteria. It is assumed that four criteria must hold for their algorithm to select a safe subset of test cases: safety, precision, efficiency and generality.

Coverage techniques aim to select only those test cases that traverse paths where the program has been changed. The coverage can lead to that some test cases are not selected, although they should have been because they traverse possibly affected parts of the program.
Like loaded systems, there are several Embedded Systems that run huge amount of software. Changes to the embedded systems though are not frequent, but will change due to modifications required either in Hardware or Software. Every Time a change is made, all the effected areas of the systems must be tested using regression testing.

In Literature several authors have contributed to different techniques using which regression testing is carried. These techniques generally are suitable for the loaded systems. An investigation is to be made to check whether the recommended regression testing approaches are suitable for testing the embedded systems. In the case that no such applicable testing methods exist for the embedded system, an architectural model is proposed and presented based on the study of the process of a feasible testing method.

With a prioritization technique, test cases can be exercised in such an order that those test cases that reveal failures early in the testing process that get identified and coverage criteria increased at a faster rate are re-executed first. It depends on the application what should be concerned when choosing prioritization criteria. It is proposed that test cases are prioritized by cost per additional coverage to reveal failures early.

[Agarwal 1993] has recommended an incremental Regression Testing method for testing the embedded systems. The incremental method proposed by Agarwal is based on simple observations.

- If a statement is not executed under a test case, it can not affect the program output for that test case.
- Not all statements in the program are executed under all test cases.
- Even if a statement is executed under a test case, it does not necessarily affect the program output for that test case.
Every statement does not necessarily affect every part of the program output.

The data in the program is represented in a control flow graph extended with information of data flow. In addition to this the changes made to program must hold the following conditions.

- The control flow graph of the program remains unchanged, i.e. no existing edges are deleted and no new ones are introduced.
- The def-sets of nodes in the program’s control flow graph remain unchanged, i.e. no changes are made to the left-hand-sides of assignment statements in the program.

There are three types of techniques in incremental regression testing which include Execution slice technique, Dynamic slice technique, and relevant slice technique.

The set of statements that are executed under a test case is called ‘execution slice’ of the program. The execution slices of all the test cases are determined during off-line processing of the program. This can be used to test a program at unit and function level. In some cases a modification to a statement needs that all test cases are selected. This happens when, for example, the predicate of a conditional statement is changed but the conditional block does not affect the output. The problem of selecting all test cases in some cases is solved by using a dynamic program slice technique. The method determines the test cases in which the modified statements affect the output. However, the method can not determine what type of modification that is made. The relevant slice technique identifies the potential dependencies of variables in an execution history. The relevant slice technique determines potential dependencies of a variable if in a path of the execution history no definitions of the variable can be found between a predicate and the use of the variable, and there exist a definition of the
variable in another path. Both paths start in the predicate and end in the computation that uses the potentially dependent variable. Sometimes the use of a variable is control dependent of a previous predicate. This can lead to unnecessary rerun of test cases. This problem is solved by excluding the statements that have control dependencies to a predicate that may affect the output from the set of statements. The relevant slice technique is the best technique among the three proposed techniques.

Incremental regression testing is suitable to loaded system and also to the embedded system. The amount of modeling to be done in this case is quite huge and takes a lot of time to conduct testing. Regression testing of embedded systems must be done at a faster rate as the solutions are low cost solutions.

[Tsai 2001] has proposed Scenario based Regression testing. He uses a test scenario model for regression testing using three perspectives which include Test Scenario Specification, Test Scenario Dependencies and Traceability to other software artifacts. Tsai proposed the following life cycle procedure for undertaking the Regression testing.

1. Test planning specifies the key tasks as well as the associated schedule and resources.
2. Test design involves design of E2E testing, including test specifications, test case generation, risk analysis, usage analysis, and scheduling tests
3. Test execution involves executing the test cases and documents the testing results
4. Result analysis analyzes the testing results, evaluates testing and performs additional testing if necessary.

He proposed regression testing using Test scenario specifications which can be designed right in the beginning of analysis stage. Test Specification is
a semi formal specification that sits between descriptive system requirements and executable test cases. Firstly, test case describes detailed test requirements in terms of data, conditions and constraints and secondly, in addition to normal inputs, test cases also capture abnormal inputs and exceptions that the system should also handle reliably.

A test scenarios specification can be presented as a template containing several sections each having an association with a set of attributes. Scenarios Description through a template is presented at Table 3.5:

<table>
<thead>
<tr>
<th>Section</th>
<th>Attributes</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>ID</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>String</td>
</tr>
<tr>
<td>Policy</td>
<td>Test Strategy</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Test Criteria</td>
<td>String</td>
</tr>
<tr>
<td>Input/Output</td>
<td>Input</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Expected Output</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td>Execution</td>
<td>Modules Involved</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Persistent Data</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Interfaces Involved</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Execution Path</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td>Conditions</td>
<td>Pre Conditions</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Post Conditions</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td>Linkages</td>
<td>Requirements</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Test Scripts</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td>Others</td>
<td>Status</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Agents</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Schedules</td>
<td>Varchar: Multi Entry</td>
</tr>
<tr>
<td></td>
<td>Risks</td>
<td>Varchar: Multi Entry</td>
</tr>
</tbody>
</table>

**Table 3.5 Template for Test Scenario Specification**

The section “General” describes identification of a test scenario with ID and Name, as well as a brief summary of the context and objective of the test scenario. The Section “Policy” defines the testing policy in terms of test
strategy and test criteria. The section "Input/output" identifies the set of data to be exercised, and the action that triggers the Execution. The Expected Output identifies the expected output data and execution delivery. The section Execution defines the set of software components that are verified by the test scenario composed of Modules, Interfaces, Persistent data storage, and Execution Path through these components. Condition defines the pre- and post-conditions for a test scenario. Pre-conditions are those factors that must be met before the execution, and Post-conditions are the expected system status after the execution. Linkage involves information for a test scenario to trace the Requirements and Test Scripts.

This template also provides other attributes, which are important for test management, including Status, Agents, Schedules and Risk. The number of test scenarios for an interconnected system can be large. For better management, test scenarios can be arranged hierarchically into a tree structure. In this way, test scenarios that share certain commonalities can be grouped together.

Dependence analysis provides the foundation for regression testing and REA (Ripple Effect Analysis). The dependencies form the perspectives of functional dependencies, input dependencies, output dependencies, input/output dependencies, persistent data dependencies, execution dependencies, condition dependencies. In addition to these dependencies when regression testing of embedded systems must be carried, the event occurrence dependencies, the pattern of event occurrences must also be considered.

Test scenarios may share either pre-conditions or post conditions, or both. Test scenarios that share a pre-condition may start executing from the same system state, triggering event, and environment setting, while test scenarios that share a post-condition may reach the same system state from different states.
The traceability enables global change impact analysis among software artifacts. Test scenarios are also traced to software elements, including subsystem components, interfaces and data, specified in execution of the template. In general, a requirement can be traced to multiple test scenarios, and a test scenario to multiple test cases (test scripts). Each test scenario may cover multiple software components along its execution path; and each software component may be tested by multiple test scenarios.

Tsai proposed that scenarios based testing be considered using program slicing. Program slicing performed on source code can be used for white-box regression testing and Ripple Effect Analysis.

REA is used to analyze and eliminate side effects due to changes and to ensure consistency and integrity after changes are made to software. It is an iterative process of change request, software modification, impacts identification and validation. It ends when there are no more ripples. The REA process is not specific to any particular programming language or design paradigm. REA keeps relationships among scenarios, software components, and requirements, no matter where the change is initiated. Whenever there is an inconsistency, REA is needed. In each iteration, impacts are identified and validated using test scenario slicing with various slicing criteria.

While the above proposed method is useful for regression testing of loaded system, it is necessary to extend the same with event based processing and also the location and method mapping for conducting comprehensive regression testing of the embedded systems. If the regression testing is to be undertaken it is necessary that a Data Model be constructed using which the changes to the basic code is recognized and the effected code is identified. Using the code, the effected events, Patterns, and the testing to be carried and the location of carrying the testing and method that should be used to carry testing can be determined.
Onama A.K. (1998) has recommended a multi-level regression testing framework. He has discussed both management and technical issues to address multi-level regression testing framework that can be easily adapted into a software development and maintenance process.

Large programs are usually developed in stages by teams of developers, testers, and managers using a development model such as waterfall, or prototyping. Each large program is decomposed into components, and each component can be further decomposed. During the process, the software is tested or inspected at various stages, like requirement stage, design stage, etc. Testing is divided into unit testing, multiple levels of integration testing, functional testing, reliability testing, usage testing, stress testing, acceptance testing, and field testing.

Regression testing should be used whenever there is any change in the software, and it should be embedded in the software development and maintenance process. It should not be an independent stage of a software development and maintenance process; instead, regression testing should be performed at each stage whenever there is a change. For example, if a module has been changed, it must be submitted to regression testing before it is submitted for integration with other modules. This is simply an application of divide-and-conquer strategy commonly used in software engineering. This approach is called Multi-Level Regression Testing (MLRT).

In MLRT, test cases may be run multiple times during the process because a test case designed for unit testing may be rerun again at integration level. This is so because at the time the concerned module is linked with other modules its faults may be detected using exactly the same test cases for unit testing. Thus, some test cases may be rerun as a quality assurance procedure. MLRT has many advantages. First, test suites can be attached to each software component at different levels of granularity. Another major
reason for practicing MLRT is that the delay in detecting faults is minimized. MLRT is time taking and is not really suitable for undertaking the testing of the embedded system. Embedded systems are event based and therefore isolation of the code to the event processing level and then undertaking the testing is important.

[Sastry, Rajasekhar et al 2007-8] have proposed regression testing of the distributed embedded systems using the WEB based service Architectures. The method proposed is modified to undertake the testing of stand alone embedded systems and the same is presented in section 3.7

3.7.1 Proposed Regression Testing Method for embedded systems

The technique proposed by [Tsai 2001] and others are not quite suitable for testing the embedded system as any change effected on an embedded systems either due to changes undertaken on either Hardware or software requires that the entire thin thread that has the changed element in it must be tested. Any change affected must be traced down the line up to identifying the thin thread.

If the thin thread that is affected is traced then all the test cases that are relevant to the thin threads can be identified and tested. Again the testing using several testing methods will be used to carry testing for carrying entire testing related to thin thread.

The data model proposed for building the semantic model will greatly help in conducting the Regression testing of the embedded systems. The Data model helps in conducting the traceability and Ripple effect Analysis. The data model used for building semantic model is presented in the figures 3.22 to Figure 3.25 to describe the algorithm proposed for conducting the regression testing of the embedded system.
Figure 3.22 present the relationships between Application tasks, Functions, Code Statements, Data Elements and function execution sequences. Fig 3.23 present the relationship between test Scenarios, complex scenarios, Thin threads, patterns, events, Test Locations and Test Methods. Fig 3.24 presents the relationships between Test Case Types, Test Commands, test Methods and Thin Threads. Fig 3.25 presents the relationships between the classes, use cases, functions, tasks, execution sequences etc.

Fig 3.22 Data Model building the code Relationships
Fig 3.23 Data Model for modeling Thin Thread Relationships

Fig 3.24 Data Model for building Test Case Relationships
Fig 3.25 Data Model for building Relationships related to Code Units

In the database the following relationships are built.

- Relationship between the simple and complex scenarios
- Relationship between the complex scenario and the thin thread
- Relationship between the code units
- Relationships between the code units and data variables
- Relationships between Tasks and the code units
- Relationship between the Data Variables of the Application System
- Relationships between the events and the thin threads
- Relationship between the thin threads and the Test method
- Relationship between the test methods and the test case types
- Relationships between the Thin threads and the patterns
- Relationship between the code units and the lines of code
- Relationship between the lines of code and the Data Variables.

The execution sequence is initiated through an event which may be an external event, or an event caused due to the passage of time. The Test pattern is associated with some input elements and output elements.

The data repository can be built from the source code. The data repository establishes relationship between the code and the data elements.
When ever changes are made by adding the data elements, the same are to be populated into the data dictionary. When modifications are made to data elements the same will be affected on the data dictionary. When data elements are deleted, the entry of data elements from the dictionary is deleted.

The Changes made to data dictionary can be recorded separately and using this data the scenarios that are associated with the data can be selected and used for the test case generation for conducting the regression testing. New scenarios may be generated and stored which will deal with the new data elements added into the system.

The changes made to the code are also stored separately and the data elements that are associated with the changed lines of code are identified. All the data variables that are indirectly associated with the data variables contained in the effected lines of code are also identified. The set of data variables that are affected are regressed over the code and the code units that are affected are identified. The Thin threads that are associated with the affected code are determined leading to determination of the affected thin threads.

The test cases that are related to the thin threads and a test method are selected and testing is carried at the locations and the methods that are related to the test scenarios/Thin Thread.

The data model will help conducting the ripple effect analysis to identify all the affected areas of the code and the data that is related to the code. The scenarios that are associated with the affected Data elements either directly or indirectly can be identified or used for generating the test cases needed for conducting regression testing.
3.7.2 Algorithm for Regression Testing of the Embedded Systems

The following algorithm helps in conducting the regression analysis of the embedded systems in a faster way.

1. Identify the functions involved in the code related to the entire application components resident on different distributed embedded systems.
2. Identify the lines of code.
3. Identify the associations of lines code with functions.
4. Identify the data variables in the code.
5. Identify the lines of code in which the data variables are involved.
6. Identify the thin threads and their association with the Tasks
7. Identify the Association of the Tasks with code units
8. Identify the Patterns and their association with thin threads
9. Identify the events and their relations with thin threads
10. Identify the test scenarios which are either basic or complex. The basic test scenarios are used to do the unit testing whereas the complex test scenarios are used to conduct either integration or system testing. Each of the Test Scenarios is identified with the input variables and the expected output variables.
11. Identify the association of test scenarios with thin threads
12. Identify the test methods that are related to the thin threads
13. Identify test case types that are related to the Test methods
14. Update the above relationships in case of either addition of new variables or new instructions.
15. Maintain the lines of code which have been changed either through addition, modification and deletion.
16. Identify the data elements that related to changed lines of code.
17. Regress the entire lines of code using the data elements involved in the changed code.
18. Identify that functions which contain the changed lines of code.
19. Identify the thin threads that are related to the affected code
20. Identify the events that are related to the thin threads
21. Generate the test cases that are related to thin thread and undertake the testing.
22. Undertake testing and compile Test Results.
23. Conduct an audit trail to assess the test coverage.

3.8 Conclusions

In this chapter, various architectural Frameworks and models are proposed which can be effectively be employed for undertaking the comprehensive testing of the embedded systems using the test case types recognized in chapter 2.0. The following Architectural Frameworks and models have been proposed in the thesis.

1. Architectural Framework for Test Environment
2. Architectural Framework for a Semantic Model
3. Architectural Framework for Process Models
4. Architectural Framework for building test scenarios
5. Architectural Frame work for END-TO-END Integration Testing
6. Architectural Framework for Regression Testing

The Architectural framework for the test environment clearly defined the environment required for testing the embedded systems comprehensively. The architectural framework required for testing the embedded system has been presented in the section 3.2

The Architectural framework for deriving the semantic model which forms the fundamental basis for comprehensive testing of the embedded systems has been presented in section 3.3. The semantic model is presented to have three blocks and the interaction between the blocks provides for interaction between various models. The relationships between all the blocks have been explained through a data model. The architecture for semantic model clearly
identified the Models that are used for conducting the Analysis, design and development of the Embedded Systems.

The models in the 1st block of semantic model provide the platform required for undertaking comprehensive testing of the embedded system. The models which include Use case, class, sequence and state charts and the relationships with Embedded Application have been presented. The data entities (Class, use case, attributes, functions, statements, events, thin threads, and tasks) required to relate all the models in the first block have also been explained.

The models presented in the 2nd block of semantic architectural framework provided the basis for identification of testing requirements, the END-TO-END integration testing requirements modeled through the definition of the Thin threads, Data models that relate the analysis, design and development inputs with thin threads, the semantic model itself, and the Regression testing model required for undertaking the regression of the embedded systems. This block also explained the relationship of test case types with thin threads and test methods. The test case generation process has also been explained.

The models presented in the 3rd block of semantic model are related to the process models required for actual testing of the embedded systems. This block also describes the test configuration required and the processes required for undertaking the test coverage and testing Analysis.

Architectural Frameworks to build the process models required for undertaking the testing at each of the locations have been presented in the section 3.4. The process models clearly identified the sequence of processing undertaken right from compilation stage to the test case generation and testing of the embedded system. All integration aspects to integrate the
Hardware, Software, test gadgets and Production system have also been presented.

The Scenario based architectural models required for building the Test case requirements from the end user perspective have been presented in the section 3.5. The building of scenarios has provided the basic framework for identifying the testing requirements from the perspective of any given application. The process of construction of the complex scenarios from simple scenarios has also been presented in this chapter. The scenarios are linked to the thin threads as the thin threads provide for END-TO-END testing of the embedded system. The scenarios are related with each other and with the Thin threads and the relations are presented in the data model. The data model presented the framework for undertaking the testing of the embedded system dynamically and using the repeatable tests when ever required.

In the section 3.6 the architectural models required for undertaking the END-TO-END (E2E) Integration testing has been presented. The pre and post conditions required to undertake the E2E has been defined through patterns. Various types of patterns that used and the required input and output parameters for E2E testing of the embedded systems have also been explained. The tree structures which are used to build the thin threads and the patterns have also been explained. The linking of the thin threads with patterns through trees has been explained. Every scenario has been mapped to a thin thread and scenarios are tested by undertaking the testing of the thin threads. The thin threads are mapped to the test methods and test case types to methods. Test case generation process for testing the thin threads has been presented.

In section 3.7 the architectural framework required for carrying the regression testing of the embedded has been presented. In this section the tracing of the changes across the entire applications and conducting of the ripple effect analysis has been presented. The process of identifying the thin
threads that are affected due to the changes undertaken has been explained. Any thin thread that has the affected components in it is proposed to be tested and this will guarantee the comprehensive testing of the embedded systems.