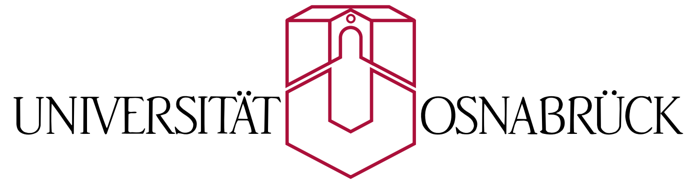



INSTITUT FÜR INFORMATIK
AG SOFTWARE ENGINEERING

# A Test Framework for Executing Model-Based Testing in Embedded Systems

*Dissertation*

zur Erlangung des Doktorgrades (Dr. rer. nat)
des Fachbereichs Mathematik/Informatik
der Universität Osnabrück

vorgelegt von

**Padma Iyenghar**

Osnabrueck,
September 2012

**Abstract**

*Model Driven Development (MDD)* and *Model Based Testing (MBT)* are gaining inroads individually for their application in embedded software engineering projects. However, their full-fledged and integrated usage in real-life embedded software engineering projects (e.g. industrially relevant examples) and executing MBT in resource constrained embedded systems (e.g. 16 bit system/64 KiByte memory) are emerging fields.

Addressing the aforementioned gaps, this thesis proposes an *integrated model-based approach* and *test framework* for executing the model-based test cases, with minimal overhead, in embedded systems. Given a chosen System Under Test (SUT) and the system design model, a test framework generation algorithm generates the necessary artifacts (i.e., the test framework) for executing the model-based test cases. The main goal of the test framework is to enable test automation and test case execution at the host computer (which executes the test harness), thereby only the test input data is executed at the target. Significant overhead involved in interpreting the test data at the target is eliminated, as the test framework makes use of a *target debugger* (communication and decoding agent) on the host and a *target monitor* (software-based runtime monitoring routine) in the embedded system. In the prototype implementation of the proposed approach, corresponding (standardized) languages such as the Unified Modeling Language (UML) and the UML Testing Profile (UTP) are used for the MDD and MBT phases respectively. The applicability of the proposed approach is demonstrated using an experimental evaluation (of the prototype) in real-life examples.

The empirical results indicate that the total time spent for executing the test cases in the target (runtime-time complexity), comprises of only the time spent to decode the test input data by the target monitor and execute it in the embedded system. Similarly, the only memory requirement in the target for executing the model-based test cases in the target is that of the software-based target monitor. A **quantitative comparison on the percentage change in the memory overhead** (runtime-memory complexity) for the existing approach and the proposed approach indicates that the **existing approach** (e.g. in a MDD/MBT tool-Rhapsody), introduces approximately **150% to 350% increase in memory overhead for executing the test cases**. On the other hand, in the proposed approach, the target monitor is independent of the number of test cases to be executed and their complexity. Hence, the percentage change in the memory overhead for the **proposed approach** shows a **declining trend w.r.t the increasing code-size** for equivalent application scenarios (approximately **17% to 2%**).

Thus, the proposed test automation approach provides the essential benefit of executing model-based tests, without downloading the test harness in the target. It is demonstrated that it is feasible to execute the test cases specified at higher abstraction levels (e.g. using UML sequence diagrams) in resource constrained embedded systems and how this may be realized using the proposed approach. Further, as the proposed runtime monitoring mechanism is time and memory-aware, the overhead parameters can be accommodated in the earlier phases of the embedded software development cycle (if necessary) and the target monitor can be included in the final production code. The aforementioned advantages highlight the scalability, applicability, reliability and superiority of the proposed approach over the existing methodologies for executing the model-based test cases in embedded systems.

# Acknowledgements

This work would not have been possible without the help and guidance of many to whom I am grateful. It is my pleasure to express my gratitude to them in my humble acknowledgement.

I am grateful to my advisor *Prof. Dr. Elke Pulvermueller* for her invaluable guidance over the past three years. Her constant support and encouragement have been vital to my progress and it has been an honor to work with her. I would like to express my gratitude to *Prof. Dr. Clemens Westerkamp*, who has made this endeavor academically possible and also co-advised my PhD. His guidance and suggestions have been invaluable to this work. For all the efforts from *Prof. Pulvermueller* and *Prof. Westerkamp*, I would like to express my deepest gratitude to both of them.

I extend my sincere thanks to *Prof. Dr. Juergen Wuebbelmann* for all the fruitful discussions I have shared with him. I also truly appreciate *Prof. Wuebbelmann* and *Prof. Westerkamp* for providing me with excellent working conditions during my stay as a research and development engineer at the Institute of Computer Engineering, UAS, Osnabrueck.

This work has been carried out in close co-operation with industrial partners *Willert Software Tools GmbH* and *BTC Embedded Systems AG*. I extend my heartiest thanks to *Eike Roemer* for the numerous fruitful discussions I have had with him. My thanks are also tendered to *Walter van der Heiden* for his interesting training sessions and discussions. I also acknowledge the interest of *Andreas Willert* in encouraging and supporting my PhD work. I extend my thanks to *Michael Spieker* for his help with measurements and numerous fruitful interactions. My special thanks to *Dr. Marc Lettrari* and *Dr. Hartmut Wittke* for their timely support and interactions during the prototype implementation involved in this thesis work.

I express my sincere thanks and appreciation to *DAAD (Deutscher Akademischer Austauschdienst, German Academic Exchange Service)*, which awarded a scholarship for my PhD work at the University of Osnabrueck. I also extend my sincere thanks to the research pool (UAS Osnabrueck) for the partial funding that I have received during my stay at UAS Osnabrueck.

It was my pleasure to have interacted with fellow researchers at international conferences. I especially would like to thank the anonymous reviewers (for the publications), whose reviews and suggestions have significantly improved this work.

I would like to dedicate this work to my *mother*. Without her endless love, encouragement and faith in me I would have not progressed this far. Special thanks to my husband *Gopal* for his support and understanding, through these years. Last but not the least, I would like to express my gratitude to my (late) *grandparents*, who taught me to get up after a fall and start again.

# Contents

**1 Introduction** **1**
1.1 Problem statement and analysis . . . 2
1.1.1 Scope . . . 3
1.2 Challenges and requirements . . . 4
1.3 Thesis outline . . . 5

**2 Background and Related Work** **7**
2.1 Background . . . 7
2.1.1 Model Driven Development (MDD) . . . 8
2.1.2 Model Based Testing (MBT) . . . 12
2.1.3 MBT and UML Testing Profile (UTP) . . . 16
2.1.4 Runtime monitoring for embedded systems . . . 19
2.2 Related work . . . 25
2.2.1 MDD for embedded systems . . . 25
2.2.2 MBT for embedded systems . . . 28
2.2.3 MBT and UTP . . . 29
2.2.4 Runtime monitoring for embedded systems . . . 31
2.2.5 Industrial evaluation scenarios . . . 33

**3 Integrated Model-Based Approach and Test Framework: An Overview** **35**
3.1 Model Driven Development phase . . . 35
3.2 Model Based Testing phase . . . 37
3.2.1 Components of the test framework . . . 40
3.3 Runtime monitoring . . . 45
3.3.1 Time and memory-aware runtime monitoring . . . 45
3.3.2 Target debugger . . . 46
3.4 Executing model-based test cases in embedded systems . . . 49

**4 Test Framework Generation** **53**
4.1 MIDI system analyzer case study . . . 53
4.2 Model Driven Development phase . . . 56
4.2.1 Software design model . . . 56
4.2.2 Detailed behavior modeling using UML state charts . . . 58
4.2.3 Mapping between formal notation and UML elements . . . 60

4.3 Model Based Testing phase . . . 62
4.3.1 Test framework . . . 62
4.3.2 Proxy Test Model . . . 63
4.3.3 Communication Interface . . . 66
4.3.4 UTP Artifact . . . 66
4.4 Algorithm . . . 68
4.4.1 Test framework generation . . . 69
4.4.2 Description of the *generateEventConsumedOnTarget* procedure . . . 74
4.4.3 State chart creation for the SUT in test framework . . . 79
4.4.4 Procedure for UTP artifact generation . . . 85
4.4.5 Communication interface creation . . . 89
4.5 Complexity Analysis . . . 98
4.5.1 Test framework generation complexity . . . 98
4.5.2 Runtime complexity . . . 103

**5 Experimental Evaluation 109**
5.1 Target debugger and target monitor . . . 110
5.1.1 Prototype implementation of the target debugger . . . 112
5.1.2 Prototype implementation of the target monitor . . . 119
5.1.3 Performance metrics . . . 128
5.2 Executing model-based test cases in the embedded system . . . 131
5.2.1 Experimental setup for the MIDI system analyzer . . . 131
5.2.2 Test case specification . . . 133
5.2.3 Test case execution . . . 138
5.3 Complexity analysis and discussion . . . 145
5.3.1 Test framework generation complexity . . . 147
5.3.2 Runtime complexity . . . 154

**6 Summary and Outlook 167**
6.1 Summary . . . 167
6.2 Outlook . . . 171

**Bibliography 173**

**Publications 181**

# List of Figures


1.1 Challenges in this thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1 MDD and MBT for embedded systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 Example of a simple UML state chart . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 Example of a sequence diagram specification at design level using a MDD tool [42] . . 11
2.4 Example of a state lifeline notation or robust notation for timing diagrams . . . . . . 11
2.5 Example of a value lifeline notation or concise notation for timing diagrams . . . . . . 11
2.6 A taxonomy of model-based testing [113] . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.7 Classification of the model-based testing process [113] . . . . . . . . . . . . . . . . . . 14
2.8 Steps involved in executing model-based test cases, offline, in embedded systems . . . 15
2.9 Steps involved in executing model-based test cases, online, in embedded systems . . . 15
2.10 General schema of black-box testing and UTP terminology . . . . . . . . . . . . . . . 18
2.11 Embedded system as a black box . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.12 Embedded system with probes for monitoring . . . . . . . . . . . . . . . . . . . . . . . 20
2.13 View of a typical hardware monitor for embedded systems . . . . . . . . . . . . . . . . 20
2.14 Possible locations of a software monitor within an embedded system . . . . . . . . . . 21
2.15 Illustration of a hybrid monitor in an embedded system . . . . . . . . . . . . . . . . . 22
2.16 Theoretical on-chip (hybrid) monitoring arrangement . . . . . . . . . . . . . . . . . . . 22
2.17 Instrumentation (memory) overhead for executing model-based testing in embedded systems using a MBT tool [43] (obtained by measurement) . . . . . . . . . . . . . . . 24

3.1 Integrated model-based approach and test framework for embedded systems . . . . . . 36
3.2 Steps involved in executing model-based test cases in the proposed approach . . . . . 38
3.3 Example of a system design model and chosen SUT . . . . . . . . . . . . . . . . . . . 41
3.4 Example showing the automatically generated test framework (based on the design model shown in Figure 3.3) with proxy classes for the chosen SUT . . . . . . . . . . . 41
3.5 Frame format for test input data sent to the target debugger by the test framework . 42
3.6 Frame format for test results sent by the target debugger to the test framework . . . . 43
3.7 Steps involved in decoding the test input data without the target debugger . . . . . . 47
3.8 Steps involved in decoding the test input data with the target debugger . . . . . . . . 47
3.9 XML file creation at the host computer . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.10 Series of steps for executing the model-based test cases in embedded systems . . . . . 50

4.1 Requirements for the MIDI system analyzer case study . . . . . . . . . . . . . . . . . . 55

4.2 MIDI protocol with a command byte for a key press action . . . . . . . . . . . . . . . 55
4.3 Simple embedded system software design model . . . . . . . . . . . . . . . . . . . . . . . 56
4.4 Message sequences in the MIDI system-design phase . . . . . . . . . . . . . . . . . . . 57
4.5 Overview of the design model for the MIDI system analyzer. The design model ($M^{App}$) is encapsulated in a package $P^{App}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.6 State chart of *PreDecoder* class ($S_1^{App(s,t,e)}$) . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.7 State chart of *MIDI_Interpreter* class ($S_2^{App(s,t,e)}$) . . . . . . . . . . . . . . . . . . . . 60
4.8 Screenshot showing the implementation of the *PreDecoder* class in a MDD tool [42] . . 61
4.9 Test framework notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.10 Test framework generated for the chosen SUT, *PreDecoder_proxy* class . . . . . . . . . 63
4.11 Screenshot of the (automatically generated) *PreDecoder_proxy* class in a MDD tool [42] 65
4.12 Communication interface component in the test framework . . . . . . . . . . . . . . . 66
4.13 Mapping between UTP terminology and formal notation for UTP artifact . . . . . . . 67
4.14 Proxy test model generated for the chosen SUT (*PreDecoder* class). (a) Proxy classes, (b) operations, (c) attributes, (d) association ends and (e) dependencies. . . . . . . . 71
4.15 An event *evValidMIDICmd(MIDI_Cmd=9855)* is generated (on the target) from the *PreDecoder* class to the *MIDI_Interpreter* class . . . . . . . . . . . . . . . . . . . . . . 74
4.16 Trace data corresponding to an event execution, i.e., *evValidMIDICmd(MIDI_Cmd=9855)* generated on the target from the *PreDecoder* class to the *MIDI_Interpreter* class . . . 75
4.17 Series of steps involved in the *generateEventConsumedOnTarget*($M_n^{App}$) procedure in the test framework generation algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.18 State chart generated automatically for the *PreDecoder_proxy* class by algorithm 2 . . 82
4.19 Series of steps involved in sending the test input data for a test case to the embedded system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
4.20 UTP artifact component created in the generated test framework for the *PreDecoder* class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.21 TCP/IP communication interface created for the chosen SUT (*PreDecoder* class) . . . 91
4.22 Functionality of the *execute()* operation in the TCP/IP communication interface . . . 93
4.23 Functionality of *decodeMsgFromTrgtDebg()* in TCP/IP communication interface . . . . 94
4.24 Steps involved in the *createComObject()* procedure to generate the decoding logic for *decodeMsgFromTrgtDebg()* function . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
4.25 Execution parameters for test framework generation . . . . . . . . . . . . . . . . . . . 99
4.26 Execution parameters for executing model-based test cases in embedded systems . . . 103

5.1 Tools, languages and example application scenarios used during prototype and experimental evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
5.2 XML file creation at the host and target monitor design at the embedded system . . . 111
5.3 An excerpt of the XML file generated (using the procedure shown in Figure 5.2) at the host computer for the MIDI system analyzer application example . . . . . . . . . . . . 113
5.4 Screenshot of the target debugger GUI. Sequence and timing diagrams shown in Figure 5.5 and Figure 5.6 respectively are visualized in the area (b) in Figure 5.4. . . . . . . . 117
5.5 Visualizing MIDI system behavior in real time using UML sequence diagram in the target debugger GUI (in the area (b) in Figure 5.4) . . . . . . . . . . . . . . . . . . . . 118

5.6 Visualizing MIDI system behavior in real time using UML timing diagram in the target debugger GUI (in the area (b) in Figure 5.4) . . . . . . . . . . . . . . . . . . . . . . 118
5.7 Target monitor frame format and parameters for various notifications . . . . . . . . . 121
5.8 Generic and JTAG based interfaces as debug communication interfaces . . . . . . . . . 125
5.9 Series of steps involved in injecting an event (test input data) from the host to the target using the target monitor (EIA-232 interface) . . . . . . . . . . . . . . . . . . . . 126
5.10 Series of steps involved in sending the trace data (e.g. test result) from the target to the host using target monitor (EIA-232 interface) . . . . . . . . . . . . . . . . . . . . 127
5.11 MIDI system overview and experimental setup . . . . . . . . . . . . . . . . . . . . . . 132
5.12 Screenshot showing an example of an UML sequence diagram test case specified manually for the MIDI system analyzer case study. Test input injected by the test driver and the expected output (messages) are highlighted within the dotted rectangle. . . . 134
5.13 Screenshot showing an example of an UML sequence diagram test case specification with state information for participating test components . . . . . . . . . . . . . . . . . 135
5.14 Screenshot showing an example of an UML sequence diagram test case specification with several sequences of events as input . . . . . . . . . . . . . . . . . . . . . . . . . . 136
5.15 Example of an UML sequence diagram test case specification with *loop* iteration operator 138
5.16 Steps involved in executing the model-based test case examples . . . . . . . . . . . . . 139
5.17 Screenshot of test case execution result summary (in .html file). This result corresponds to the test case specified in Figure 5.12 . . . . . . . . . . . . . . . . . . . . . . . . . . 142
5.18 Screenshot of test case execution result with state information in the test framework. This result corresponds to the test case specified in Figure 5.13 . . . . . . . . . . . . . 143
5.19 Screenshot showing the status of the steps in a test case execution. This example shows a *pass* result. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
5.20 Screenshot showing the status of the steps in a test case execution. This example shows a *fail* result. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
5.21 Overall view of the mapping of the various components in the test framework to the steps involved in executing the test cases using the proposed approach . . . . . . . . . 146
5.22 Functionality of a stopwatch embedded software application . . . . . . . . . . . . . . . 148
5.23 Mounted spark detector [33] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
5.24 Overall view of the design model for the spark extinguishing system example . . . . . 150
5.25 Steps for determining the time required for sending the test input data between the test framework and the target debugger ( $Tcp_{send}$) . . . . . . . . . . . . . . . . . . . 155
5.26 (a) Test case execution result in the target debugger GUI (corresponds to the test case/ test input data in Figure 5.12) and (b) visualizing the same scenario by pressing a key on the key board (envisaged functionality), for the MIDI system analyzer . . . . . . . 160
5.27 Memory requirement for the test framework $[A_{sut}{\cdot}S_{TF}] + [n_{tc}{\cdot}n_{dr}]$ on the host computer with increasing number of test cases $n_{tc}$ . . . . . . . . . . . . . . . . . . . . . . . . 163
5.28 Memory overhead in an embedded system for existing vs proposed approach . . . . . . 165

6.1 Proposed approach and contributions in this thesis . . . . . . . . . . . . . . . . . . . . 168

# List of Tables


2.1 UTP concept groups and stereotypes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.2 Tool support for MDD (subset) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3.1 UTP concept groups and stereotypes (concise subset) . . . . . . . . . . . . . . . . . . . 44

4.1 Mapping between the formal notation and UML elements for the system design model 62
4.2 Mapping between the formal notation and UML elements for the test framework . . . 64
4.3 Mapping of the trace data frame format to an example (in Figure 4.16) . . . . . . . . 95

5.1 Operations supported by the *Monitor* routine in the target monitor . . . . . . . . . . . 120
5.2 Trace data frame format for target monitor notifications . . . . . . . . . . . . . . . . . . 122
5.3 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
5.4 Target monitor memory requirement for different debug interfaces (Cortex-M3) . . . . 128
5.5 Time spent in the monitor routine for an event consumed notification (Cortex-M3) . 130
5.6 Empirical results for test framework generation complexity for various scenarios . . . . 151
5.7 Parameters in time-complexity $\Gamma_r^T$ (runtime) for the test case specified in Figure 5.12 . 156

# List of Algorithms

1 : Test framework generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
2 : State chart creation for the SUT in the test framework . . . . . . . . . . . . . . . . . . 81
3 : UTP artifact creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4 : Communication interface creation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

# Acronyms

| | |
|---|---|
| API | Application Programming Interface |
| GUI | Graphical User Interface |
| ISR | Interrupt Service Routine |
| LED | Light Emitting Device |
| MARTE | Modeling and Analysis of Real-Time and Embedded Systems |
| MBT | Model Based Testing |
| MDA | Model Driven Architecture |
| MDD | Model Driven Development |
| MIDI | Musical Instrument Digital Interface |
| MMI | Man-Machine Interface |
| OCL | Object Constraint Language |
| OMG | Object Management Group |
| PIM | Platform Independent Model |
| PIT | Platform Independent Test |
| PSM | Platform Specific Model |
| PST | Platform Specific Test |
| QA | Quality Assurance |
| RAM | Random Access Memory |
| ROM | Read Only Memory |
| RTA | Real-Time Agent |
| RTOS | Real-Time Operating System |
| RXF | Real-Time Execution Framework |
| SPT | Schedulability Performance and Time |
| SUT | System Under Test |
| SysML | Systems Modeling Language |
| UART | Universal Asynchronous Receiver Transmitter |
| UI | User Interface |
| UML | Unified Modeling Language |
| UTP | UML Testing Profile |
| XML | Extensible Markup Language |

# Chapter 1

# Introduction

Nowadays embedded systems are being increasingly used in a wide variety of application scenarios. Some examples of embedded systems in our day-to-day lives and industrial fields are cardiac pacemakers, brakes in cars, smart home appliances, avionic applications, robot controllers and wind mill control systems. In most of these devices, embedded systems often contain hardware components in conjunction with software routines. However, embedded software development is different from desktop programming. For instance when developing PC-based software, memory can be treated as infinite (and free) and that there will always be enough CPU power. On the other hand, with the embedded software one needs to be more circumspect and parsimonious in resource usage. This is attributed to the critical memory-size and real time constraints that the embedded system must often adhere to. These stringent constraints vary based on the type of the processor (4/8/16/32 bit) used in the microcontroller of the embedded device and the functions it has to perform. This implies that a majority of the embedded systems will comprise of software of varying size and complexity.

Classical software development and testing methodologies such as the V-Model approach [6] have been successfully applied for several years in the embedded systems domain. However such methods are reaching their limits in the context of embedded systems, where the time-to-market needs to be shortened drastically in order to meet economic requirements. The manifold constraints in embedded software development such as the limited resources (e.g. memory), stringent real time requirements, growing variety and complexity pose a big challenge for embedded software engineering. These mandatory requirements have to be addressed both during the development as well as in the testing phases in embedded software engineering projects. In order to achieve this goal, structured embedded software engineering and automation are inevitable. It is therefore essential that the development of embedded systems take advantage of methodologies such as the Model Driven Architecture (MDA) that enable higher levels of software productivity.

Over the last few decades the software industry has gone through several paradigm shifts. This is evident in the tradition of the evolution of programming languages from the assembly language over structural programming languages to the object oriented paradigm. These shifts have followed a pattern of moving to higher level of abstractions. This is driven by the demands for building more complex systems and the development of tools that let software engineers focus on a higher level of abstraction. In this series of evolution in software engineering, the MDA introduced by the Object Management Group (OMG) [79], is considered as the next paradigm shift. The system modeling

and test modeling techniques based on MDA are collectively termed as Model Driven Development (MDD) and Model-based testing (MBT) respectively. In these methodologies the implementation level source code or the test harness (e.g. C language) are obtained using model transformation and automatic code generation techniques based on models specified at a higher abstraction level (e.g. using Unified Modeling Language (UML)). In the recent past, the principles of MDA (i.e., MDD and MBT) are applied not only for the development of desktop-based systems but also for embedded software systems.

In the following section, the applicability of MDD and MBT for embedded systems is briefly outlined. An analysis of the drawbacks in applying the MDA paradigm for embedded systems is provided. This gives a perspective on a few aspects/criteria that a model-based approach should meet, such that it is applicable for resource constrained embedded systems and industrially relevant examples. Based on this, the scope and objectives of this thesis are derived. In line with the scope of the thesis, the challenges and contributions of this thesis are outlined.

## 1.1 Problem statement and analysis

While MDD and MBT are gaining inroads individually for their applicability in embedded software engineering projects, their full-fledged and integrated usage in real-life embedded software engineering projects is still an emerging technology. This can be attributed to the following main drawbacks.

- Lack of methodologies using corresponding modeling languages for MDD and MBT phases. An example of employing corresponding modeling languages for MDD and MBT phases is the usage of UML for MDD and UML, UML Testing Profile (UTP) for MBT.
- Dearth of experimental evaluation of MDD/MBT approaches in real-life embedded software engineering projects.
- Further, while there are several research directions for MDD and MBT, a framework for executing the model-based test cases in resource constrained embedded systems has not been sufficiently explored. An example of a resource constrained embedded system is a 16 bit system with 64 KiByte[1] memory.

As a consequence, a majority of the MBT techniques used in embedded software engineering projects execute the model-based test cases at the host computer, rather than the embedded system itself. On the other hand, commercially available model-based tools such as [43] support executing the model-based test cases in embedded systems. Such tools make use of techniques such as source code instrumentation, which involves adding or rewriting source code/binaries in memory, in order to incorporate and execute the test code. However, the overhead introduced by such techniques is significant on the embedded system. For example, the memory requirement on the target for executing the model-based test cases in comparison[2] with the application source code increases by approximately 150% to 250% [45] in a MBT tool [43]. This is the case for simple application scenarios comprising of 2, 4 and 8 classes respectively in the design model. Clearly, it is intuitive to perceive that such tools/methodologies are not suitable for executing the model-based test cases in complex application

[1] 1 KiByte=1024 bytes
[2] Obtained by measurement [45]

scenarios involving resource constrained embedded systems. Moreover, the overhead arising from such techniques could also influence the real time behavior of the embedded system. Furthermore, these aspects could influence the reliability, scalability and applicability of such approaches for resource constrained embedded systems. They also pose a hurdle in applying MBT for industrially relevant examples.

Hence, in order that a model-based test automation approach for executing MBT is applicable in real-life embedded software engineering projects involving resource constrained embedded system scenarios, it is imperative that it meets the following criteria, namely:

- Use of complementary modeling languages/domains for MDD and MBT phases. This would also help in reducing the tools training and the overall project costs.
- Predictable complexity measures (e.g. time and memory) both on the host computer and on the embedded system. With measurable and pre-determined complexity measures the approach can be termed as time and memory-aware. This can also pave the way for minimizing the influence on the real time characteristics of the embedded system.
- Scalable across application scenarios of varying size and complexity.

However, the lack of robust tools/methodologies which adhere to the aforementioned criteria, is a hurdle in applying the MBT approach(es) for executing model-based testing in resource constrained embedded systems and also industrially relevant examples.

### 1.1.1 Scope

Addressing the aforementioned gaps, the main scope of this thesis is to propose an integrated model-based approach and a test framework for executing the model-based test cases in resource constrained embedded systems using a less/minimally intrusive testing methodology. In this context, the overall goal of this thesis can be stated as follows:

**To develop an approach for model-based test automation and test case execution using high-level test cases (e.g. specified with UML sequence diagrams) in resource constrained embedded systems.**

This thesis addresses the following aspects, which can be derived from the above overall goal.

- Can an integrated model-based approach, that uses corresponding modeling languages for MDD and MBT phases, be developed for test automation in resource constrained embedded systems? An example of employing corresponding modeling languages for MDD and MBT phases is using UML for MDD and UML, UTP for MBT. An example of a resource constrained embedded system is a 16 bit system with 64 KiByte RAM/ROM.
- How can models be reused from the design model to an automatically generated test framework, wherein the test framework automates the test case execution process?
- What are the necessary artifacts on the embedded system and the host computer that should support the automatically generated test framework for executing the high-level model-based test cases in resource constrained embedded systems? In the context of this thesis, "high-level

model-based test cases" refers to test cases specified with UML diagrams, such as an UML sequence diagram.

- What are the complexity measures (e.g. time and memory) both on the host computer and the embedded system. Are they predictable and measurable beforehand? (ie., are they time and memory-aware?)
- How will the trade-offs arising out of the proposed approach influence the execution of model-based test cases in real-life software engineering projects involving resource constrained embedded systems?

An important point to note is that, this thesis does not deal with automatic model-based test case generation [9]. In turn, this thesis concentrates on automatic test framework generation for executing the model-based test cases in resource constrained embedded systems.

## 1.2 Challenges and requirements

In line with the problems and the scope outlined above, this thesis proposes a test automation approach for executing the model-based test cases in resource constrained embedded systems. The challenges faced and the resulting contributions can be grouped into the following four main areas, namely:

**(1) Integrated model-based approach:** Surveys on MBT approaches [19], [77] indicate that the existing MBT approaches are usually not integrated with the software development process. It is intuitive to perceive that it is beneficial to minimize the use of tools and modeling languages for the individual phases of MDD and MBT. This can be achieved, for example, by employing corresponding modeling languages for MDD and MBT phases (e.g. using UML for MDD and UML, UTP for MBT). In line with the scope of this thesis, this aspect is especially advantageous for specifying the test artifacts in a corresponding modeling language in the MBT phase to provide model-based infrastructure support for executing the model-based test cases in embedded systems. However, such comprehensive approaches are currently in the nascent stages in both industry and academia [19].

**(2) Generic test framework:** Several MBT approaches found in the literature are restricted in terms of their application to a specific modeling language. A generic approach, (i.e., its functionality and application is not restricted to a specific modeling language), towards generating a model-based test framework at the host computer is missing. By using such a test framework, the test cases can be executed at the host computer and only the test stimuli/test input data can be executed in the embedded system. This approach provides an opportunity for the tester to execute the high-level model-based test cases in resource constrained embedded systems, without introducing significant overhead in the embedded system (as in the existing approaches). In addition, using the test framework the test results can be visualized as models (e.g. UML sequence diagrams) in real time, at the design level. However, such approaches are missing in the existing methodologies and tools for embedded systems.

**(3) Runtime/online software monitoring:** While MBT is widely applied for embedded software engineering projects, the model-based test cases are, often, not executed on the target platform.

Instead most of the test case execution process is carried out on the host computer. In some scenarios, tools such as [43] are used for executing the model-based test cases in the target platform. However, the overhead introduced by such tools, which employ extensive source code instrumentation, renders these tools unusable for resource constrained embedded systems. Hence, there arises a need for a generic, less-intrusive or a minimally intrusive monitoring methodology for executing the model-based test cases in the embedded system. The complexity measures both on the host computer and the embedded system, while employing such a methodology, should be measurable beforehand. Nevertheless such a methodology for executing the high-level model-based test cases in resource constrained embedded systems is missing in the existing approaches.

**(4) Real-life embedded software engineering project example:** Though numerous MBT approaches are proposed in the literature, most of them are not evaluated in a real-life embedded software engineering project. However, to demonstrate the relevance, applicability and scalability of a MBT approach, experimental evaluation in real-life embedded software engineering examples is desirable.

The aforementioned goals/challenges are denominated as *challenges* in the forthcoming paragraphs. The structure of this thesis is outlined in the following section, with an emphasis on the aforementioned four main challenges.

## 1.3 Thesis outline

The thesis is organized as follows: This chapter gives an overview of the research topic of this thesis. It introduces the problems this work is dealing with and the objectives of this thesis. A brief outline of the challenges faced is provided by grouping them into four main areas. Figure 1.1 provides an outline of the challenges addressed in this thesis.

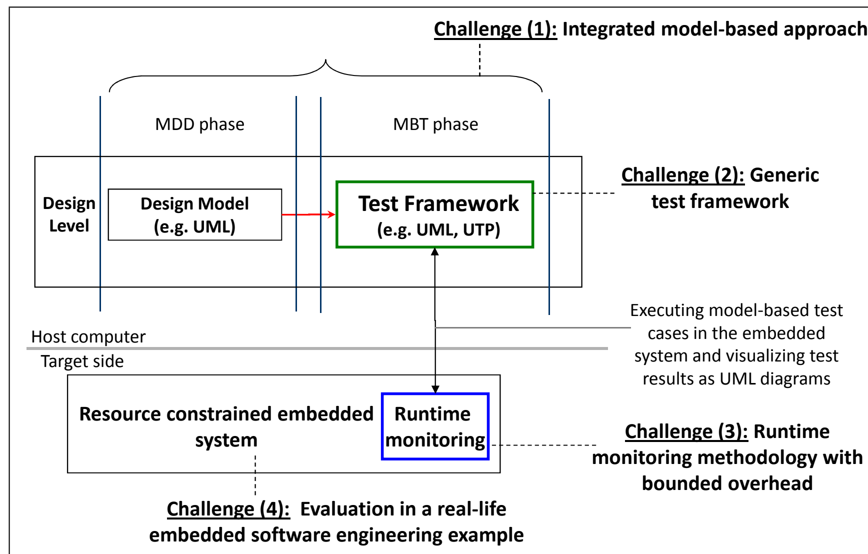



Figure 1.1: Challenges in this thesis

**Chapter 2** is sub-divided into two main sections, which discuss the background and related work

pertaining to the challenges (1), (2), (3) and (4). In section 2.1 a background on embedded systems, and the principles of MDA, i.e., MDD and MBT are provided. Advantages concerning the development of an integrated model-based approach using corresponding modeling languages in MDD and MBT phases are outlined. Runtime monitoring approaches for embedded systems and their application in the context of executing the model-based test cases in the embedded system are discussed.

In section 2.2, related research work pertaining to the challenges outlined in chapter 1 is provided. Section 2.2 begins with a discussion on the related work regarding the application of MDD, MBT for embedded systems. Next, related research work and tool support for executing the model-based test cases in embedded systems are discussed. Finally, a brief outline of the embedded software application scenario in which the existing MBT approaches are applied is provided.

**Chapter 3** provides a general overview on the proposed integrated model-based approach and test framework for embedded systems, in line with the challenges (1), (2) and (3). An overview of the MDD and MBT phases in the proposed approach is provided in this chapter, wherein corresponding modeling languages are applied for both these phases. This discussion results in shaping the components of the generic test framework proposed in this thesis. In addition, a generic software-based runtime monitoring approach, used in conjunction with the test framework, is outlined in this chapter.

**Chapter 4**, provides an in depth discussion of the test framework approach outlined in the previous chapter, thereby relating to the challenges (1), (2), (3) and (4). A generic notation for the system design model and the test framework is introduced in this chapter. The notation is explained in detail based on a real-life embedded software engineering project example comprising of a Musical Instrument Digital Interface (MIDI) system analyzer case study. A test framework generation algorithm which generates the necessary test artifacts based on a chosen System Under Test (SUT) and a given system design model is proposed in this chapter. The various steps of the algorithm are described using the MIDI system analyzer example. A complexity analysis of the proposed approach, discussed in two perspectives, provides insights into the overhead parameters in the proposed approach.

In **chapter 5**, an experimental evaluation of the components of the test framework and their performance metrics are discussed. Executing the model-based test cases using the proposed approach is illustrated using example scenarios. A detailed empirical evaluation of the parameters in the complexity analysis of the proposed approach, both during the test framework generation and runtime, is dealt with in this chapter. The quality metrics are obtained by an experimental evaluation of a prototype implementation of the proposed approach in (three) real-life embedded software engineering project examples (challenge (4)).

**Chapter 6** completes this work with a summary and outlook. The capabilities and limitations of the proposed approach are reviewed. The challenges outlined in this thesis are recalled and influences of the contributions of this thesis are outlined. Some aspects for future work are highlighted.

# Chapter 2

# Background and Related Work

This chapter is sub-divided into two main sections (2.1 and 2.2), which discuss the background and related work of this thesis respectively. In section 2.1, a background on embedded systems and the principles of MDA, i.e., MDD and MBT is provided (challenge (1)). A taxonomy of MBT and the classification of the MBT process is outlined. Advantages concerning the development of an integrated model-based approach using corresponding modeling languages in MDD and MBT phases are discussed (challenges (1) and (2)). In addition, a runtime monitoring approach for embedded systems and its application in the context of executing the model-based test cases in the embedded system are discussed. Based on this the need for a less/minimally intrusive monitoring methodology is perceived (challenge (3)).

Section 2.2 begins with a discussion on the related work regarding the application of MDD and MBT for embedded systems. Modeling languages for both the phases and usage of corresponding modeling languages in MDD and MBT phases are discussed (challenge (1)). Next, the methodology used for executing the model-based test cases in embedded systems is elaborated by providing an overview of the related research work and tool support in this area (challenges (2) and (3)). Finally, a brief outline of the embedded software application scenario in which the existing MBT approaches are applied is provided (challenge (4)).

## 2.1 Background

An embedded system can be coarsely defined as a computer system that is dedicated to specific tasks [8]. It is often subject to limited resources (e.g. memory, power), critical real time constraints and may make use of a Real-Time Operating System (RTOS). This is in contrast with the desktop systems, which are generic computing platforms. Embedded systems often contain hardware components in conjunction with software routines. However, embedded software development is different from desktop programming because of the aforementioned manifold constraints in embedded systems. Also, the implications of software failure is much more severe in embedded systems than in desktop systems.

In today's world, embedded systems are omnipresent and enormous in number. A majority of the embedded systems comprise of software of varying size and complexity. The need for new embedded software development and testing methodologies arises from the increasing complexity and sophistication of embedded systems [55]. Towards this direction, the model-based paradigm is widely used to

assuage the difficulties arising in the design, development and testing of embedded software systems. In this context, MDA introduced by the OMG [79] has been gaining rapid attention in the recent decade. New and automated ways of software modeling, development and testing based on MDA such as MDD and MBT are currently explored for embedded software engineering projects. Figure 2.1 provides an illustration of the steps involved in MDD and MBT phases.

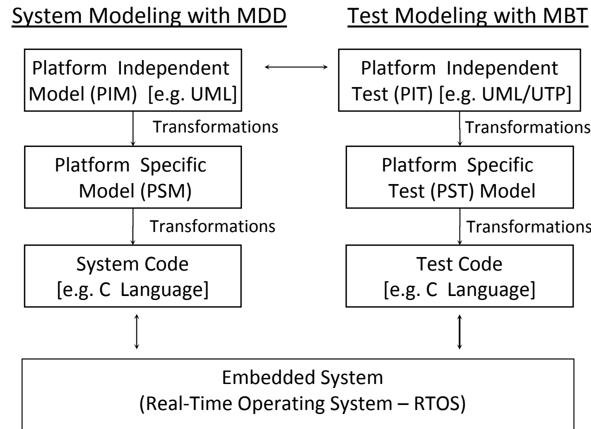



Figure 2.1: MDD and MBT for embedded systems

### 2.1.1 Model Driven Development (MDD)

For system modeling using MDA, initially the Platform Independent Model (PIM) is described at a higher abstraction level (Figure 2.1). Based on the PIM and using platform specific input, the Platform Specific Model (PSM) is obtained by automatic model transformation. From PSM, the implementation level source code is then obtained using automatic code generation techniques. This is collectively termed as MDD (Figure 2.1). This implies that it is now easier to validate, correct and implement on different platforms, thereby enabling easy integration and inter-operability across different systems. This aims to mitigate the software development time, aid in producing good quality software and increase productivity.

MDD is related to multiple standards including modeling languages such as UML [112] and Systems Modeling Language (SysML) [99]. UML is a widely used general-purpose modeling language in the field of software engineering. It is primarily used for software/system modeling [9]. Not only general UML diagrams are used for modeling embedded systems, but also UML profiles introduced by the OMG, which are new stereotypes specific for the real time and embedded systems domain. In short, during the last few years, UML has become the *lingua franca* among system modelers all over the world [63]. Apart from general UML diagrams, the UML profiles introduced by the OMG consist of a set of new stereotypes for a particular domain [112]. For instance, the UML profile for Schedulability, Performance and Time (SPT) and Modeling and Analysis of Real time and Embedded systems (MARTE) are OMG specifications targeting the real time and embedded domain [79]. From this, we see the widespread support for UML as a modeling language for embedded systems.

In this thesis, UML is used for embedded software modeling by employing various diagram types such as class diagrams, state charts and sequence diagrams. Adhering to the MDD methodology, the

system code (to be executed in the embedded system) is obtained using automatic code generation methodologies.

UML state machines or state charts can be defined as an enhanced realization of system behavior, as a composition of finite number of states. Often embedded software systems are event-driven, which means they continuously wait for the occurrence of some external or internal event. Therefore, in this thesis UML state charts are used to represent the detailed reactive behavior of the individual classes. An overview on the UML state charts is provided in section 2.1.1.a.

Further, in this thesis, a novel methodology to visualize the behavior of the embedded system, executing the system code, is outlined [50] [49]. UML diagram types such as sequence and timing diagrams are used to create visual models of the embedded system behavior in real time at the design level/host computer. The UML sequence diagrams are also used to visualize the test results (e.g. using MBT) for embedded systems at the host computer/design level using the proposed test framework. Hence, a brief description about UML sequence and timing diagrams is provided in section 2.1.1.b.

#### 2.1.1.a Detailed behavior modeling using UML state charts

Detailed modeling of the behavior or functionality of a given class can be represented by means of UML state chart diagrams, the semantics of which is described below. UML state machines or state charts can be defined as an enhanced realization of system behavior, as a composition of a finite number of states. The behavior is in turn represented as a series of events that could occur in one or more possible states. Thereby, each diagram usually represents objects or instances of a single class/module and track the different states of its object through the system. Thus, the UML state charts support actions that depend on both the state of the system and the triggering event, entry and exit actions (associated with states). Often embedded software systems are event-driven, which means they continuously wait for the occurrence of some external or internal event. Once the incoming event is recognized, such systems react by performing the appropriate action which may include manipulating the hardware components or generating events that trigger other internal software components. Once the system handling is complete, the system goes back to a waiting (state) for the next event.

UML state chart diagrams often comprise of an *initial/default transition* which originates from a solid circle and specifies the default state when the system first begins. This transition, which is not labeled, is not triggered by an event. The default transition ends in a default state in the state chart. A state chart can comprise of one more more states. A state captures the relevant aspects of the system behavior and comprises of optional *entry* and *exit actions*. These actions are executed upon entry to a state or upon exit from a state respectively. Each state comprises of one or more transitions triggered by an *event*. Each *transition*, triggered by an event (trigger), consists of one or more *guard* conditions and a respective *action*. The guard conditions are boolean expressions evaluated dynamically based on the value of the extended state variables and event parameters. Guard conditions affect the behavior of a state machine by enabling actions or transitions only when they evaluate to TRUE and disabling them when they evaluate to FALSE. Activities such as changing a variable, performing I/O, invoking a function or changing to another state (state transition) etc are examples of actions. Hence, each action comprises of details such as the action itself, its *source* and *destination*.

An event, once generated, goes through a processing life cycle consisting of up to three stages. First, an event is *received* where it is accepted and awaits processing (e.g. in an event queue). Next, the event instance is *dispatched* to the state machine at which point it becomes the current event.

Finally, it is *consumed* when the state machine finishes processing the event instance.

An example of a simple UML state chart, to switch (i.e., to toggle) an LED between On and Off states, is shown in Figure 2.2. This state chart comprises of two states namely *Off* and *On*. In this

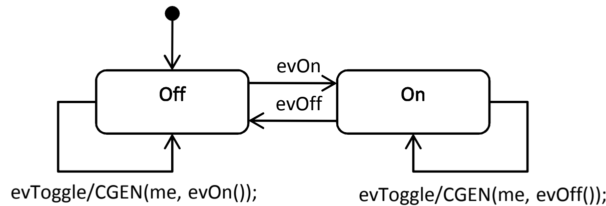



Figure 2.2: Example of a simple UML state chart

state chart, the default transition (which originates from a solid circle) is shown for the *Off* state, as the *Off* state is considered as the default state at which the system first begins. On triggered by an event *evToggle*, the *Off* and *On* states invoke functions *evOn()* and *evOff()* respectively. This is indicated, for example, by the statement *evToggle/CGEN*[1] *(me, evOn())* and *evToggle/CGEN(me, evOff())*. These functions *evOn()* and *evOff()* are used to turn *On* or turn *Off* an LED. On reception of the event *evOn* (in the *Off* state) or *evOff* (in the *On* state), a transition to the respective other state (i.e., *On* or *Off* state) occurs, as seen in Figure 2.2.

#### 2.1.1.b UML sequence and timing diagram

In the UML 2.x versions, sequence and timing diagrams are classified under the set of interaction diagrams. These diagrams emphasize the flow of data/control among the entities modeled in the system and depict the interactions among themselves.

The sequence diagram shows how objects communicate with each other in terms of sequence of messages. It also indicates the life spans of objects relative to those messages [20], [63]. In sequence diagram, time increases from top to bottom and the spacing between the messages are not relevant. Sequence diagrams are used to show specific scenarios or interactions with sequenced exchange of messages. These messages can be events, operation/function calls, state changes, instance creation/ destruction, etc. The sequence diagram consists of object lifelines with messages between them representing interactions. There are annotations such as status messages or constraints such as timing requirements. An example of a sequence diagram specification at the design level (using a MDD tool [42]), showing the sequence of interactions during an automatic car wash is illustrated in Figure 2.3. This example comprises of four object lifelines namely, *Customer*, *PreWash*, *MainWash* and *Drying*. A loop interaction operator which repeats the sequence of operations over a specified number of iterations (e.g using guard parameter $cars.carNr \leq 5$), is also illustrated. The time interval over which each operation is expected to be finished is specified as < 1 min> (*PreWash*), < 3 min> *MainWash* and < 1 min> (*Drying*).

The timing diagram is a specific type of interaction diagram where the focus is on timing constraints. The differences between a timing diagram and a sequence diagram are the reversed axes so that the time is increased from left to right and the lifelines are shown in separate compartments

[1]The macro *CGEN(destination, event(parameters))* is used to generate an event as per the rules of the modeling tool Rhapsody [42]. This tool is also used in the evaluated prototype and examples.

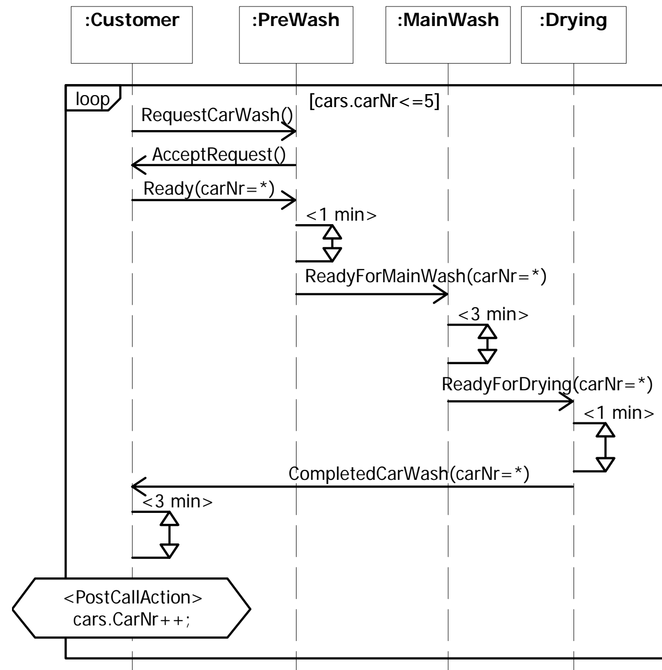



Figure 2.3: Example of a sequence diagram specification at design level using a MDD tool [42]

arranged vertically [108]. The horizontal axis displays elapsed time in ticks or in other chosen time units, while the vertical axis can be labeled with a given list of states or values, i.e., there can be a state lifeline or a value lifeline in the vertical axis. A value lifeline shows the change of value of an item over time, whereas the state lifeline shows the change of state of an item over time. This is also termed as the concise notation and the robust notation respectively.

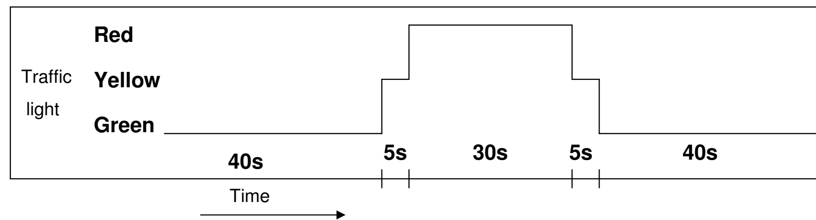



Figure 2.4: Example of a state lifeline notation or robust notation for timing diagrams

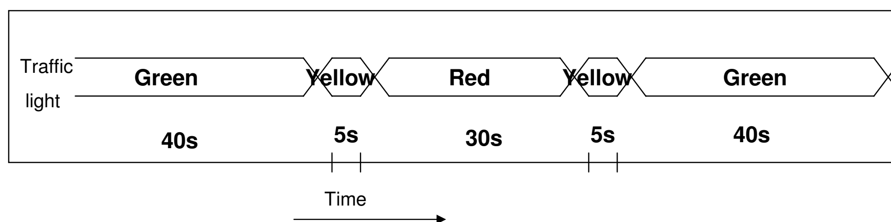



Figure 2.5: Example of a value lifeline notation or concise notation for timing diagrams

Consider a simple traffic light example, with green, yellow and red lights with the condition that

exactly one of these lights should be switched on at a time and the sequence repeated forever. The duration constraint shall be green light: 40 seconds, yellow light: 5 seconds, red light: 30 seconds and yellow light: 5 seconds. Modeling at the design level using the robust and concise notation from UML timing diagram, for the above example, is shown in Figure 2.4 and Figure 2.5 respectively.

From the characteristics of the sequence and timing diagrams described above, it is intuitive to judge the applicability and usefulness of these interaction diagrams in the field of real time embedded software systems with timing constraints. For example, it can be used in design of control software for fuel injection system in an automobile, where a MDD approach is applied.

### 2.1.2 Model Based Testing (MBT)

MBT deals with the application of the model-based approach for designing and executing the necessary artifacts to perform software testing. Applying a philosophy similar to that of MDD (Figure 2.1) for test modeling, once the system design model is defined at the PIM level, a Platform Independent Test (PIT) model can be derived (right side of Figure 2.1). This model can be transformed directly to test code or a Platform Specific Test (PST) model. After each transformation step, the test design model can be refined and enriched with test specific properties. Finally, the test model can be transformed into executable test code [123]. In other words, while employing MBT methodology a test designer needs to only create an abstract model of the SUT and then the MBT tool generates a set of test cases from that model. By this, the overall test design time is reduced. An added advantage is that one can generate a variety of test suites from the same model simply by using different test selection criteria. For instance, the various model-based testing approaches are classified into a taxonomy, in [113], as shown in Figure 2.6.

The dimensions are clustered into different groups according to whether they affect the model, the test generation process or the execution of the generated tests. As seen in Figure 2.6, there exists several methodologies for developing test models and for test case generation. These are also applicable for embedded software engineering projects. However, the test execution option, i.e., methodology to execute the test cases in the target system needs further exploration. This is, in particular, true for embedded software engineering projects. To gain further understanding, consider the MBT process itself in detail. It is classified into the following steps [113] as shown in Figure 2.7.

(a) Modeling the SUT and/or its environment

(b) Generating abstract tests from the model

(c) Concretizing the abstract tests to make them executable

(d) Executing the tests on the SUT and assigning verdicts

(e) Analyzing the test results

The first step of the MBT process (step (a) in Figure 2.7) involves creating an abstract model of the system that needs to be tested. Requirements document and the test plan document are some of the typical inputs for this step (highlighted using a dotted line in Figure 2.7). The model is created with only the certain key aspects pertaining to testing the SUT, thereby omitting detailed specification of the SUT. The model can be specified in various modeling languages such as the UML and also annotated with requirement identifiers. The second step (i.e., step (b) in Figure 2.7) in MBT deals

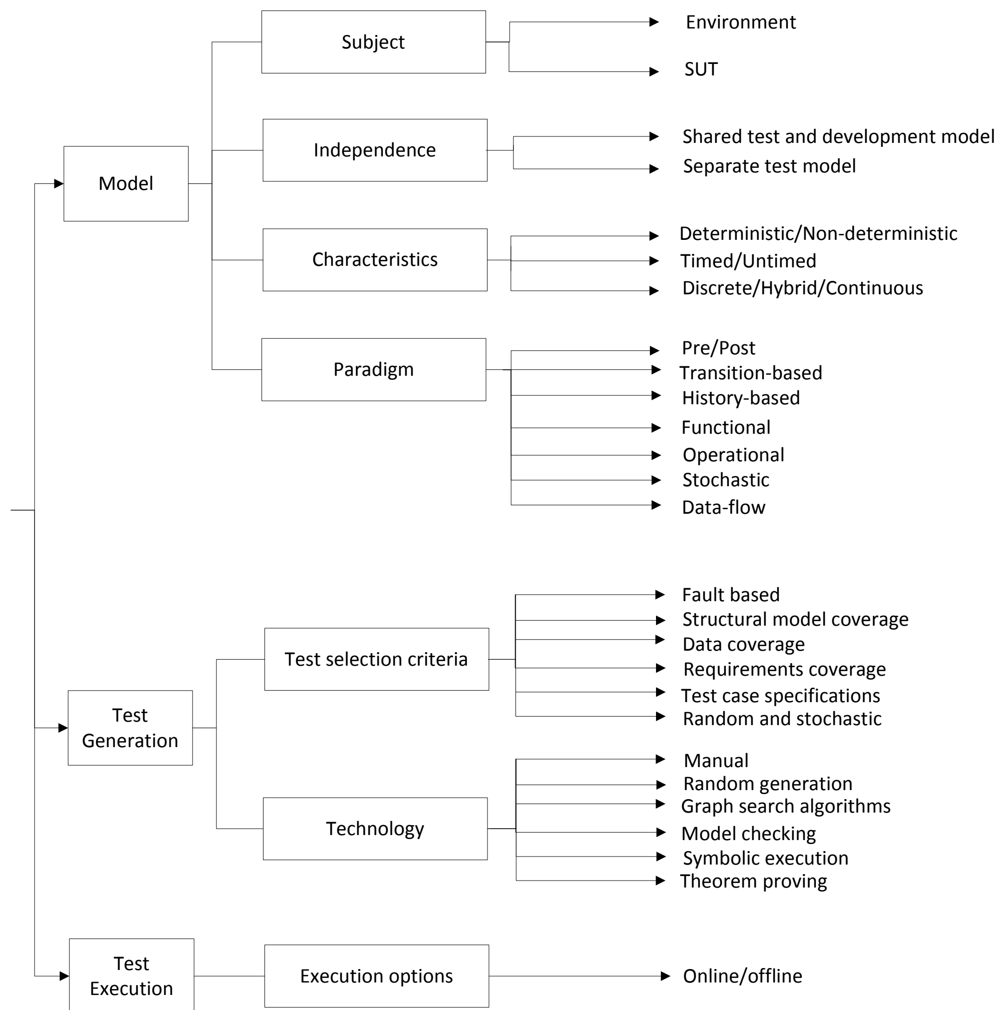



Figure 2.6: A taxonomy of model-based testing [113]

with generating abstract tests from the model, based on certain test selection criteria provided by the end user. The third step ((i.e., step (c) in Figure 2.7)) is to transform the abstract tests into executable concrete tests, for instance using a transformation tool. The fourth step (step (d) in Figure 2.7) deals with executing the model-based test cases, obtained so far, on the SUT. The fifth step (step (e) in Figure 2.7) in the MBT process deals with analyzing the test results. Corrective actions based on the results such as determining the fault that caused the failure are taken during this step. Note that a typical set of tools used during these steps, namely a test case generator tool, a test script generator tool and a test execution tool are highlighted in Figure 2.7.

### Offline/Online execution

Most often in embedded software engineering projects, the model-based test cases are executed at the host computer and not directly on the embedded system. Model-based test cases can be executed in the embedded system, in two ways namely offline and online testing. While existing tools/techniques [30], [43] support offline testing of embedded systems, executing model-based test cases (online) on

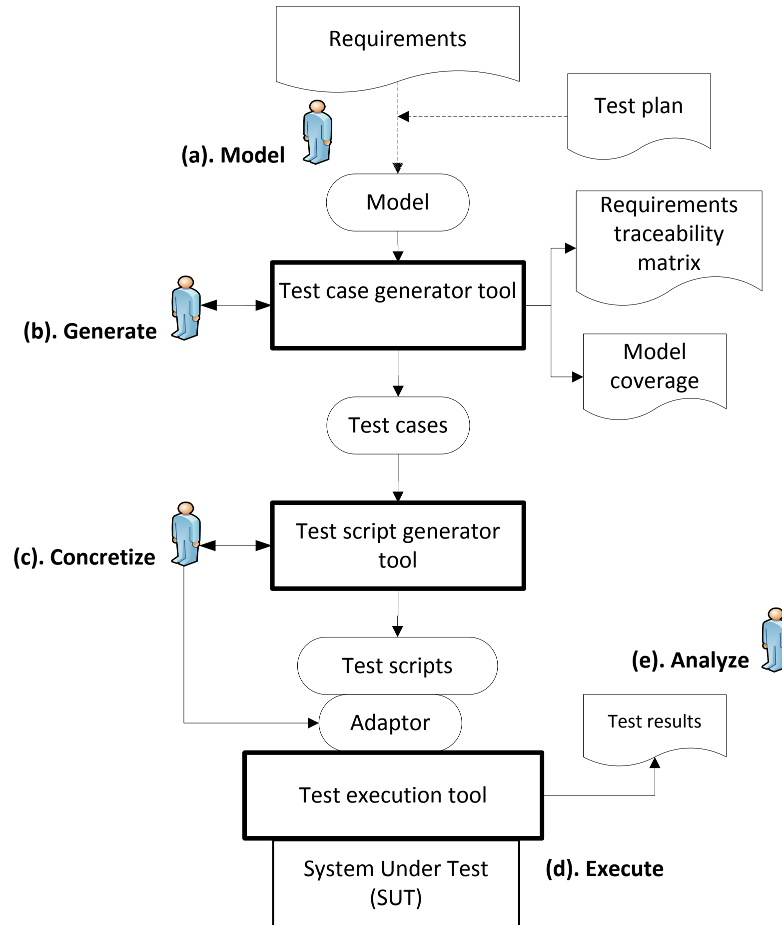



Figure 2.7: Classification of the model-based testing process [113]

the embedded system is not supported.

In this thesis, online and offline execution modes are defined as follows. In offline MBT, once the test cases are generated by the MBT tool, the test cases are executed on the SUT. The results are then analyzed after the test case execution process is completed. Online MBT means that the MBT tool connects "directly" to a SUT, tests it dynamically and analyzes the test results. A typical series of steps involved in executing model-based test cases in the embedded system, offline, in the existing tools is illustrated in Figure 2.8.

In the first step (Figure 2.8), for a given application scenario, the required test harness is generated from the set of model-based test cases. The test harness thus varies based on the number of test cases and also the application scenario. In the second step (step 2 in Figure 2.8) the test harness is downloaded directly on the embedded device. The third step involves executing the test cases in the embedded system. Next, the results for the test case execution process are computed. In the next step, the results are stored in intermediary formats (e.g. text file, XML file) on the host computer. Once the test cases are executed and the results are stored in respective files, the test case execution process can be considered complete. Finally, the test results are read from the respective files and interpreted as pass or fail. The main disadvantage here is that the test harness required for executing the test cases is downloaded on the embedded system and the test case execution process is carried

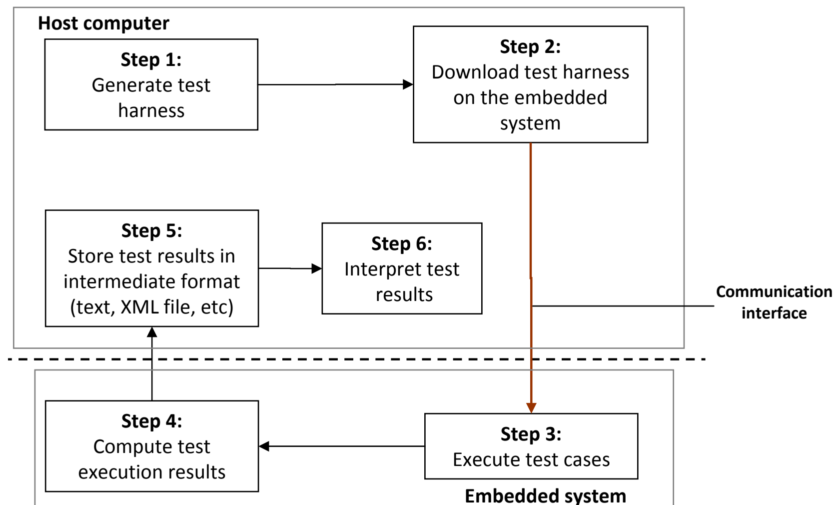



Figure 2.8: Steps involved in executing model-based test cases, offline, in embedded systems

out in the embedded system. The test harness varies based on the complexity of the application under consideration. However, the embedded system may not have the required memory to accommodate the test harness. This is particularly true for resource constrained embedded system scenarios.

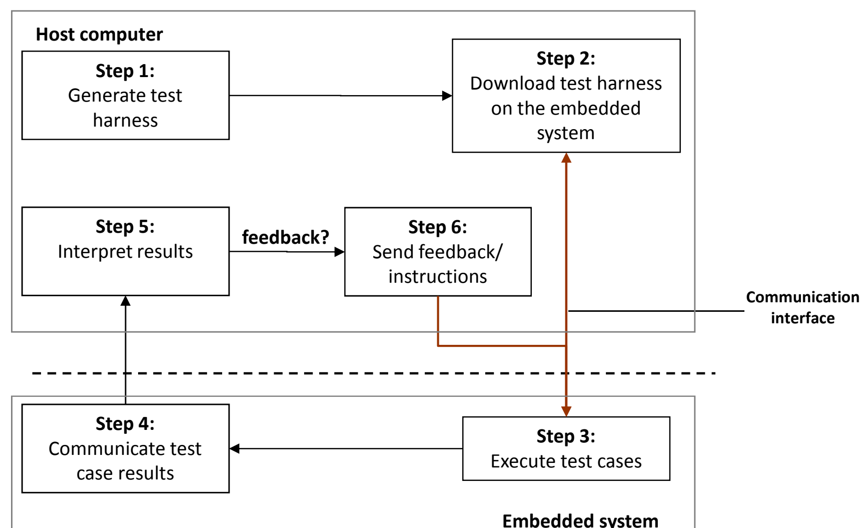



Figure 2.9: Steps involved in executing model-based test cases, online, in embedded systems

An example scenario illustrating the steps involved in executing the model-based test cases, online, in the embedded system, using the existing tools is shown in Figure 2.9. Here, the test harness is downloaded on the embedded system and the test case is executed in the embedded system. However, during the test case execution, there exists a direct communication link between the embedded system and the host computer. The test results can be interpreted at the host computer. Depending on the interpreted test results, any feedback (if required) can be sent to the target and the next step of execution can be determined. An advantage of online testing is that based on the available test results, the next step of the test case execution process can be determined. However, in addition to

the disadvantages outlined for offline MBT, here the overhead involved in sending the test data back and forth between the embedded system and the host computer can influence the real time properties of the application/system code.

**Mapping to challenges (1) and (2)**

A description of the MDD and MBT methodologies provided so far relates to challenges (1) and (2) introduced in chapter 1. This pertains to the development of an integrated model-based approach and test framework using corresponding modeling languages for MDD and MBT phases, especially in the context of an infrastructure support for executing the model-based test cases in embedded systems. Towards this direction, UML can be considered as a widely used modeling language for the MDD phase. In line with the MBT steps illustrated in Figure 2.7, several MBT approaches discussed in the literature concentrate on generating the test cases using models [19], [77]. This corresponds to step (b) in Figure 2.7. The test models used for test case generation in step (b), for instance, are derived from the system design model with/without the involvement of input such as requirements document or test plan document. Often, the test models are specified using UML diagrams such as state charts, sequence diagrams and use case diagrams. In some cases, these diagrams are also annotated with constraints, for example specified in Object Constraint Language (OCL) [19], [9]. Test models can also be specified as finite state machines [20], timed automata, simulink models [68] and in other languages in the context of modeling reactive/embedded systems such as Esterel [24]. However, these pertain to specification of the test model used for generating the test cases, i.e., step (b) in Figure 2.7.

A few approaches discussed in the literature concentrate on steps (a), (c) and (e) in Figure 2.7 [9]. On the other hand, the methodology involved in executing the test cases obtained from MBT, i.e., step (d), for embedded systems is not discussed [77], [19]. Further, an infrastructure to handle the test data generated in the test models and executing model-based testing (step (d)) especially in resource constrained embedded systems is missing in the existing tools and approaches.

Moreover, concrete steps for the definition of a corresponding modeling language for MBT phase, which includes a specification of infrastructure support for executing MBT in the SUT, has not been undertaken until the introduction of the UTP in the year 2005. On the other hand, Testing and Test Control Notation (TTCN-3) can be considered as an international standardized language for testing. However, TTCN-3 has its roots predominantly in the area of testing hardware and software components of IT and telecommunication systems.

Therefore, it can be stated that prior to the introduction of UTP, alternatives to UTP in the form of concrete/standardized modeling languages, especially corresponding to UML, or profiles does not exist. This thesis aims to explore the applicability of UTP as a corresponding modeling language for the specification of test infrastructure support, for executing the model-based test cases in embedded systems. Therefore, a background on the UTP, thereby relating to challenges (1) and (2), is provided in the following section.

### 2.1.3 MBT and UML Testing Profile (UTP)

In order to support UML with testing related activities (i.e. in the MBT phase), the UTP standard has been introduced by the OMG in the year 2005. Fundamentally, the UTP provides a basis for systematic testing and integration particularly in UML-based development environments. The UTP

provides concepts which target the development of test models for black-box testing where the internal events of the SUT are kept hidden. The profile is based on UML 2.0 and extends its meta-model by the stereotype extensibility mechanism [6], [123]. The UTP allows capturing all the needed information for black-box test approaches to evaluate the correctness of system implementations. The UTP can be used stand-alone for handling the test artifacts or in an integrated manner with UML for handling the system and test artifacts together. The UTP standard [111] introduces four concept groups covering the following aspects: *test architecture*, *test behavior*, *test data* and *test time*, as shown in Table 2.1.

Table 2.1: UTP concept groups and stereotypes

| **Test Architecture** | **Test Behavior** | **Test Data** | **Test Time** |
|---|---|---|---|
| SUT | Test objective | Data pool | Timer |
| Test component | Test case | Data partition | Time zone |
| Test context | Defaults | Data selector | |
| Test configuration | Validation action | Wild cards | |
| Test control | Verdicts | Coding rules | |
| Arbiter | | | |
| Scheduler | | | |

The *test architecture* group provides a set of concepts to specify the structural aspects of a test context covering the test components, the SUT, their configuration, etc. The *test behavior* group provides concepts to specify test behavior, their objectives and the evaluation of the SUT. The *test data* concept group is used to specify data used as stimuli (to the SUT). The time constraints and observations can be specified using concepts in the *test time* group in UTP. Together, these concepts define a modeling language for visualizing, specifying, analyzing, constructing and documenting the artifacts of a test system [6], [79].

This thesis deals with the automatic generation of a "test framework infrastructure" and the architectural support at the host side for executing model-based testing in embedded systems. In the test framework, the UTP concept group elements that provide support to describe the structural aspects of the components involved in the test and their relationships are chosen. Hence, among the extensive set seen in Table 2.1, the elements from the *test architecture* concept group is chosen for the approach proposed in this thesis. The following section describes the individual elements in the *test architecture* concept group, as defined in [6], [111], [123].

**Test architecture concept group:** The test architecture concept group contains the concepts needed to describe the elements involved in the test and their relationships. The elements in this group are described below with the help of Figure 2.10, which provides a representation for the general schema for black-box testing. Consider the SUT as a black box as illustrated in the general schema of black-box testing in Figure 2.10. The ***SUT*** can be accessed only by operations via public interfaces. The test system consisting of ***test components*** interacts with the SUT. It is assumed that the communication connections among the test components and the SUT is configured before executing the tests. In the UTP, the test component is considered as an active object within a test system performing the test behavior. Tests are performed in a certain context. The ***test context*** concept in UTP consists of test information like *test cases* that belongs to the behavior aspect of testing, *test configuration* which reflects the test environment as well as *test control*. In a ***test configuration***, the

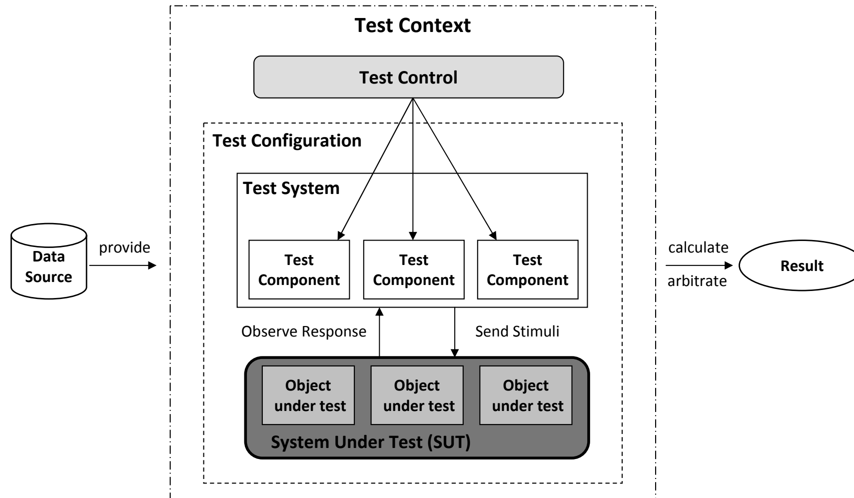



Figure 2.10: General schema of black-box testing and UTP terminology

initial setup of the test components and the connection to the SUT is specified. All test cases listed in the same test context share the same test configuration. A ***test control*** specifies the invocation of the test component behavior within a test context. The behavior of a test component is specified in test cases. The decisions about test case executions are made in the test control specification.

A test must deliver its results in a certain form to the external world. In case of UTP, they are in the form of *verdicts*. The evaluation of the verdicts is performed by a component in the test system called the ***arbiter***, to calculate the final test result. Arbitration can be defined as an evaluation process to calculate the final test result derived from local results of the test components in a test system. To perform this, an arbitration strategy is provided by the arbiter. In UTP, the arbiter interface provides interface operations in order to get and set the test verdict. To get a verdict, a test component provides its temporary local verdict to the arbiter. To set a verdict, the arbiter is invoked by a validation action and the test verdict stored in the arbiter is updated. According to the UTP standard, every test context should have an implementation of the arbiter interface. In summary, an arbiter is an entity within a test context which is capable of determining the final verdict of a test case based on the input from the test components realizing the test case. If there is no special arbiter interface implemented, a default arbiter like the one recommended in the UTP specification must be taken into account. In practice, the arbitrated verdict is returned implicitly and not shown in the test case behavior. In the prototype implementation of the proposed approach (discussed in chapter 4), the default arbitration strategy provided by the tool vendor, based on the UTP [111], is used.

In UTP, a ***scheduler*** is provided to control the creation and destruction of the test components since the test components can be created and destroyed at any time during the test execution. It also maintains the existence of each component and is aware of the participation of each test components in all the test cases. A scheduler collaborates with the arbiter and informs it when to issue the final verdict. Further features concerning verdicts are dealt within the UTP test behavior concepts. The SUT is stimulated and observed by the test system which gets its test data from data sources which may be a pool of data, partitions of data with disjoint data-sets or real time data generated by sensors or detectors. In UTP this part is specially dealt with and refined in the UTP test data concepts.

For a distributed test system, test components are assigned to *time zones* in order to achieve time-synchronization among themselves. This is treated in UTP time concepts. In summary, a *scheduler* is an entity within a test context that controls the running of the test cases. It keeps track of the creation and destruction of the test components and gives instructions to the existing test components when to start executing a test case. It communicates with the *arbiter* when the time is right to produce a verdict for a test case.

A common methodology for making use of an UML-based profile is to implement the profile as a set of stereotypes (e.g. in a MDD/MBT tool), which can be included in a given project. Then, the required/specific elements of the profile (e.g. UTP) can be made use of for a given UML element. This can be achieved by specifying the UML element with a particular stereotype from the profile. In this thesis, this methodology is followed for the specification and definition of the UTP elements.

In summary, to develop an integrated model-based approach and a generic test framework, relating to challenges (1) and (2), this thesis envisages the use of corresponding modeling languages for the MDD and MBT phases. In this context, this thesis makes use of (a) UML for specifying the design model during MDD phase and (b) UML and UTP artifacts (in the proposed test framework) for executing the model-based test cases in the embedded system, during the MBT phase. Towards this direction, an outline of the steps in MDD and MBT phases has been provided, in this section, along with a background on the relevant artifacts from UML and UTP for the two phases.

### 2.1.4 Runtime monitoring for embedded systems

In this section, a background on runtime monitoring approaches for embedded systems is provided. This relates to challenge (3) which pertains to runtime monitoring methodologies used for applications such as executing the model-based test cases in embedded systems. Given this background, the need for a less/minimally intrusive monitoring mechanism is perceived in this section.

Despite the evolution of improved software development and testing practices, possibilities of software failure in the "real world" still persists for end-users [116]. To observe the software execution in its final environment, techniques for monitoring software behavior and performance against specified rules have been developed. Observing and examining the behavior of software execution at runtime can be termed as runtime or online software monitoring and verification [116]. Software monitoring has been in use for over 35 years for a variety of domains and application purposes. Some of the domains in which runtime monitoring is applied include, distributed systems, fault-tolerant systems, real time critical systems and embedded systems.

Domains such as the embedded systems present particular challenges for monitoring, as the system internals may not be easily observable and have limited resources or real time constraints. Figure 2.11 shows an example of executing tests on an embedded system considering it as a black box. On the other hand, Figure 2.12 shows monitoring the embedded system behavior with the help of probes, in addition to the general test process. Thus, to observe a target system elements of a monitoring system such as probes are attached to the system (or placed within it) in order to provide information about the system's internal operation. As seen in Figure 2.12, probes provide an intermediate output from the system in addition to its end output allowing the system to be seen as more than a black box. Such probes can be actual hardware probes that monitor internal system signals. Alternatively, some code which performs monitoring can also be added to the target software to observe the internal

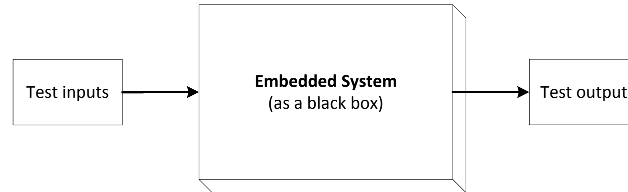



Figure 2.11: Embedded system as a black box

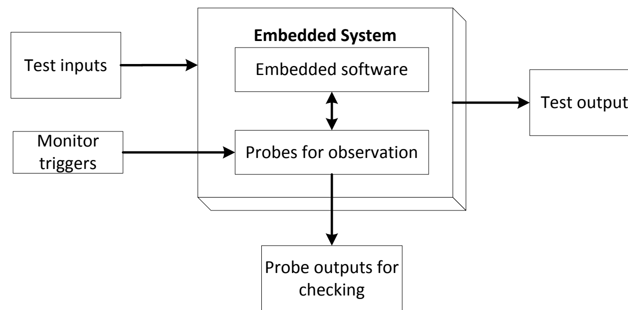



Figure 2.12: Embedded system with probes for monitoring

operations of the embedded software program. This process is termed as *instrumentation* and the code used for monitoring can be called *software probes*. Based on the aforementioned techniques, runtime monitoring of embedded systems, in general, can be classified into hardware, software, hybrid and on-chip monitoring systems.

### Hardware monitoring

Hardware monitoring deals with the use of internal system signals with hardware probes, in the embedded system, to observe the behavior of the target system (Figure 2.13). As shown in Figure

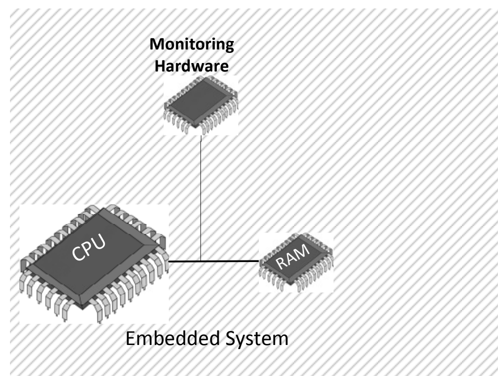



Figure 2.13: View of a typical hardware monitor for embedded systems

2.13, a dedicated monitoring hardware is attached to the embedded system. The observed data is sent by the monitoring hardware for verification, external to the embedded system. A major advantage of using a hardware monitoring approach is its non-intrusive feature. A disadvantage of a hardware monitoring approach is the inapplicability of hardware monitors for more complex systems

and problems in newer systems offering less physical probe points.

### Software monitoring

In a software monitoring approach, additional code termed as "software probe" is added to the target's system software to examine the behavior of the embedded system. As seen in Figure 2.14, in software

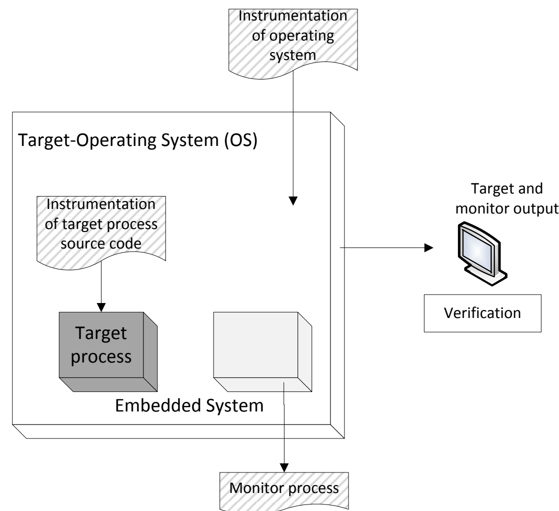



Figure 2.14: Possible locations of a software monitor within an embedded system

monitoring there are different ways how the target system can be modified through the addition of software for monitoring. For instance, software probes can be added to the target application code itself (instrumentation) or can be added to the operating system of the embedded system or it can be implemented in a separate monitor process. Software monitors added to the target application code, can be included either statically or dynamically.

### Hybrid monitoring

Hybrid monitoring refers to approaches that use a combination of additional software and hardware to monitor an embedded system. Figure 2.15 shows an example of a hybrid monitoring approach for embedded systems. Hybrid monitoring relies on the advantages of each approach and at the same time attempting to mitigate their disadvantages. Thus, by taking advantage of hardware monitoring features, hybrid monitoring is likely to be a less intrusive methodology. On the other hand, since complex systems may not have much support for physical probes, hybrid monitoring also relies on additional monitoring code to be executed in the embedded system. For example in Figure 2.15, the source code of the target process (i.e., the process being monitored) is instrumented. Additional code (instrumentation point) is added to it to emit events when certain process features being monitored are updated. These events are sent to the dedicated monitoring hardware. This hardware analyzes the events and checks them against certain pre-defined monitoring rules. The event analyzer provides the dedicated monitor output.

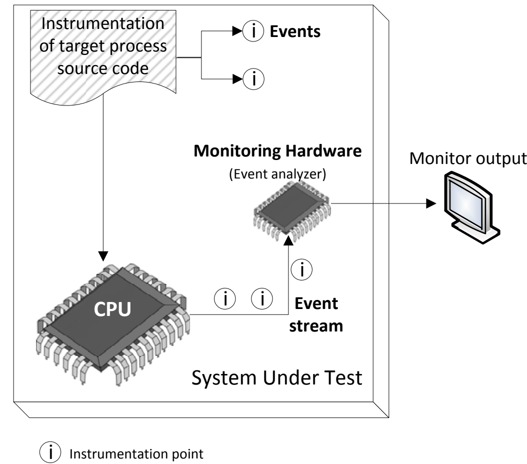



Figure 2.15: Illustration of a hybrid monitor in an embedded system

### On-chip monitoring

On chip monitoring is a type of non-intrusive monitoring methodology facilitated through the use of additional on-chip hardware. In this technique, the overhead of communicating observations to external checking mechanisms can be placed upon a small amount of dedicated hardware within the embedded system. An advantage of on-chip monitoring is that non-intrusive monitoring is facilitated through the use of additional on-chip hardware. This is also referred to as the real time trace functionality [29] for embedded systems. A pre-requisite is that the microcontroller in the embedded system must support this feature. However, while employing such techniques, there is a need for additional hardware interfaces allowing the observations to be communicated to the external monitors. On-chip monitoring mechanisms and their applicability are discussed in [94], [115], [66]. An illustration of a theoretical on-chip (hybrid) monitoring arrangement is shown in Figure 2.16.

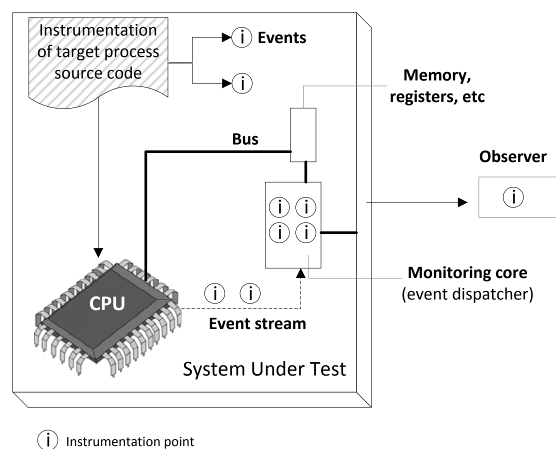



Figure 2.16: Theoretical on-chip (hybrid) monitoring arrangement

Based on the classification of the monitoring approaches for embedded systems, introduced above, the pros and cons of each monitoring approach can be perceived and summarized as follows.

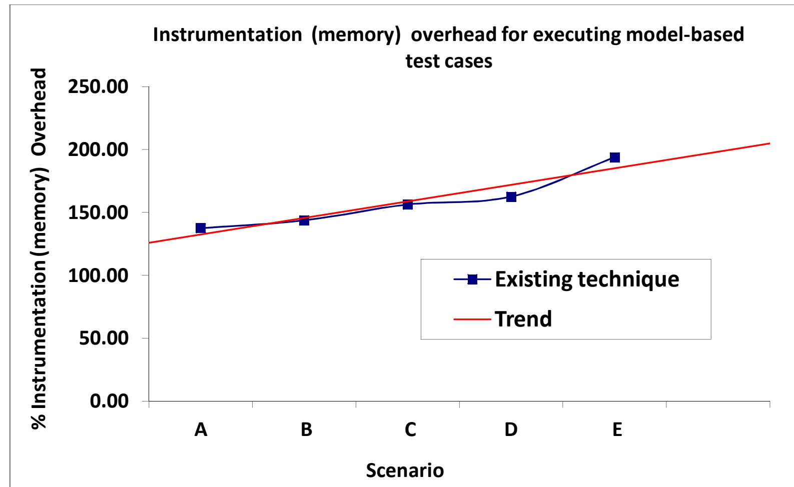
- An advantage of a hardware monitoring approach is its non-intrusive feature. However the disadvantage is the inapplicability of hardware monitors for more complex and new systems offering lesser physical probe points.
- Software monitoring can be considered as a generic approach (e.g. independent of the hardware) which provides flexibility for the tester to insert instrumented code on the target at desired locations. However, the instrumented code may introduce significant overhead (memory, time) in the embedded system.
- Hybrid monitoring relies on the advantage of both hardware and software monitoring approach, while attempting to mitigate their disadvantages.
- On-chip monitoring can be considered as the enabling technology for the future, with its non-intrusive features. However, the microcontroller in the embedded device must support this feature.

The (four) classifications of runtime monitoring approach discussed above can be used for a variety of application purposes in embedded systems. This depends on the applicability of the monitoring approach (its pros and cons) for the end-user's requirements. Potential applications of the runtime monitoring methodology for embedded systems include profiling, performance analysis, software optimization, software fault-detection, diagnosis and recovery, debugging, verification and testing. The following section concentrates on the background for application of runtime monitoring in the context of executing the model-based test cases in the embedded system. This relates to challenge (3) introduced in chapter 1.

**Runtime monitoring for executing model-based test cases in embedded systems**

As mentioned in chapter 1, there are some commercially available tools for executing the model-based test cases in the embedded system. However, the instrumentation overhead introduced by such tools is significantly high on the embedded system. Let us consider a MBT tool Rhapsody-Test Conductor [43], which enables model-based testing of embedded systems. This tool supports the specification of model-based test cases and executing model-based test cases in embedded systems. The test harness generated for the test cases is downloaded on the target and executed on the target. This is realized by the use of software monitoring and adding instrumented code to the application source code. However, the amount of test harness required for executing the test cases in the embedded system increases with an increase in the application code size. To gain a deeper understanding, a simple experiment is performed to determine the instrumentation overhead introduced by the MBT tool, Rhapsody Test Conductor [43]. Figure 2.17 shows the percentage increase in the instrumentation (memory) overhead w.r.t to the application code size.

The instrumentation (memory) overhead can be coarsely defined as the amount of additional code required to execute the test cases in the embedded system. The test harness needs to be downloaded on the target and then the test cases are executed. The four application scenarios in Figure 2.17 comprise of design models of increasing size and complexity. The trend in the instrumentation overhead indicates that there is an increase in the instrumentation overhead with an increase in the application code size. For instance, for a simple application scenario comprising of 2 classes (scenario A), the existing methodology (e.g. in tools such as [43]) requires approximately 150% increase in the test harness




Scenario A, B, C, D, E comprise of 2, 4, 5, 6, 8 classes respectively in design model

Figure 2.17: Instrumentation (memory) overhead for executing model-based testing in embedded systems using a MBT tool [43] (obtained by measurement)

(w.r.t the application code size) to be downloaded in the embedded system. Such high overhead could also influence the real time characteristics of the embedded system. From this, it is clear that the existing methodologies (in MBT tools (e.g. [43])) involves highly intrusive testing techniques and downloading enormous test harness on the embedded system for executing the model-based test cases. Such tools may be applied for testing embedded systems which are not subject to stringent real time requirements and are not critically limited in terms of computing resources. On the other hand, it is intuitive to perceive that, such techniques are not suitable for resource constrained embedded systems (e.g. 16 bit system/64KiByte memory).

In general, one must be aware of the intrusive testing methodologies and unbounded delays that various tools may introduce while executing the model-based tests in the embedded system. This aspect gains significance for resource constrained embedded systems. Thus, there arises a need for a less/minimally intrusive monitoring methodology, for executing the model-based test cases in the embedded system. This relates to challenge (3) introduced in chapter 1.

**Less/minimally intrusive monitoring methodology for executing the model-based test cases**

Choosing a probing mechanism that is suitable for runtime monitoring of embedded systems presents some challenges. For instance, the probes must be capable of observing enough information about the internal operation of the system to fulfil the purpose of the monitoring. Obviously, the observations required are dependant on the purpose of the monitoring. Next, the addition of such probes to the target system should not affect its normal behavior. The predicament in this case is that the constraints on resource usage hinder the use of software probes. However, the probes are likely to provide significantly high overhead, which leads to interference with the normal operation on the target system. High overhead could compromise core system resources or affect scheduling, causing interference to the application running in the embedded system [116]. These factors necessitate a less-intrusive or a minimally intrusive monitoring methodology for embedded systems. This is especially

true in the case of resource constrained embedded systems.

Relating to challenges (2) and (3), this thesis proposes a generic test framework which makes use of a runtime monitoring methodology for executing the model-based test cases in the embedded system. The test framework and its components intend to reduce, as far as possible, the overhead involved in executing the model-based test cases in the embedded system. Towards this direction, a generic, software-based monitoring methodology is proposed and developed (as a prototype) in this thesis. The software-based, runtime monitoring methodology is intended to be a static/constant, optimized software routine, i.e., independent of the (number) of test cases to be executed and their complexity. With the pre-determined and bounded overhead parameters for the monitoring routine, the monitoring routine can remain in the final production code.

By using such a test framework, the test cases can be executed at the host computer and only the test input data can be executed in the embedded system. The monitoring methodology in the embedded system aids in inserting the test input data and execute it in the embedded system. This approach provides an opportunity for the tester to execute the model-based test cases on resource constrained embedded systems, without introducing significant overhead in the embedded system (as in the case of existing tools). Thus, the key idea is that the test framework uses the aforementioned software-based, runtime monitoring methodology to insert the test input data in the embedded system, while the test framework takes care of the test case execution process and test automation (at the host computer). Such a methodology is especially beneficial for executing the model-based test cases in resource constrained embedded systems.

So far, in this section, the background pertaining to challenges (1), (2) and (3) have been provided. In section 2.2, related work pertaining to challenges (1), (2), (3) and (4) are discussed.

## 2.2 Related work

This section begins with a discussion on the related research work and tool support regarding the application of MDD and MBT for embedded systems. Modeling languages for both the phases and the usage of corresponding modeling languages in MDD and MBT phases are discussed (challenge (1)). Next, related research work and tool support for executing the model-based test cases in embedded systems is discussed. This gives ideas of the advantages and pitfalls in the existing approaches. Based on this, a mapping to the goals/challenges of the thesis can be obtained. The methodology used for executing the model-based test cases in embedded systems is discussed by providing an overview of the the related work in this area. Thus the related work mapping to challenges (2) and (3) is elaborated. Finally, a brief outline of the embedded software application scenario, in which the existing MBT approaches are applied, is provided. This gives a background on the related work concerning challenge (4), which pertains to an empirical evaluation of the test automation approach in real-life embedded software engineering projects.

### 2.2.1 MDD for embedded systems

The challenge (1) introduced in this thesis deals with the development of an integrated model-based approach and usage of corresponding modeling languages for MDD and MBT phases. In line with challenge (1), related work and tool support pertaining to MDD for embedded systems is discussed

in this section.

Classical software development and testing methodologies such as the V-Model approach [6] have been successfully applied for several years in the embedded systems domain. The need for new embedded software development and testing methodologies arises from the increasing complexity and sophistication of embedded systems. The main advantage from the MDD methodology is that, by making use of models for the specification of system requirements, followed by automatic code generation from the design model, it is easier to validate, correct and implement on different platforms. This enables easy integration and inter-operability across different systems [55]. Studies conducted in [12], [26], [54], [55], [59], [93] indicate that the MDD approach promises several advantages and superiority superseding the traditional ways of developing embedded software. A major advantage cited in these studies is that the model-based paradigm aims to mitigate software development time, aids in producing good quality software and increases productivity.

MDD is related to multiple standards including the UML and SysML. In [93], a model driven approach using SysML for multi-core embedded processors is presented. It claims that an application framework for developing multi-core embedded software using an MDA approach with automatic code generation from SysML models reduces the workload and makes it easy to develop embedded software. Similarly, a case study involving a model based approach and component oriented development for embedded systems is presented in [12]. It concludes that using a MDD approach for embedded systems, enables adaptable applications with higher-than-normal reuse rates.

A study on modeling embedded software using general-purpose UML is carried out in [21], [56], [86], [90]. Apart from general UML diagrams, the UML profiles introduced by the OMG are also used for modeling embedded systems. The UML profiles consist of a set of new stereotypes for a particular domain. For instance, the UML profile for Schedulability, Performance and Time (SPT) and Modeling and Analysis of Real-Time and Embedded systems (MARTE) are OMG specifications targeting the real time and embedded domain [79]. Studies based on embedded software development using the aforementioned profiles are available in [17], [34], [75]. From these we see the widespread applicability of UML as a modeling language for embedded systems.

Some of the tools currently available for supporting the MDD phase, in general, are [1], [11], [23], [42], [67], [68], [80], [96] and [108] (Table 2.2). Features supported in these tools include modeling using UML, code generation with/without platform specific inputs, support for the implementation of a specific UML-profile, etc. Commercially available tools such as [23], [42] also support MDD for embedded systems.

In line with challenge (1) which pertains to the usage of corresponding modeling languages in MDD and MBT phases, in this thesis, UML is used for embedded software modeling by employing various diagram types. Adhering to the MDD methodology, the system code to be executed in the embedded system is obtained using automatic code generation methodologies, supported in MDD tools. As per the challenges outlined in the introduction, the main focus of this thesis is proposing an integrated model-based approach and test framework for embedded systems. The test framework and its components aid in executing the model-based test cases in the embedded system without introducing significant overhead in the embedded system. In addition, a novel methodology to visualize the behavior of the application software executed on embedded system using UML sequence and timing diagrams, is briefly touched upon. Hence, related research work and tool support pertaining to visualizing the embedded software behavior in real time, using UML interaction diagrams (sequence

Table 2.2: Tool support for MDD (subset)

| Tool | Modeling Notation | MDD Phase | | |
|---|---|---|---|---|
| | | Specification | Code generation | Code generation languages |
| Matlab [68] | Simulink, Stateflow | + | + | C, C++ |
| Enterprise architect [23] | UML | + | + | C, C++, Java, C# |
| BridgePoint [11] | UML | + | + | C, C++ |
| Papyrus [80] | UML | + | + | C, C++, Java |
| Rhapsody [42] | UML, SysML | + | + | C, C++, Java, Ada |
| StarUML [96] | UML | + | + | C++, Java, C# |
| MERAPI modeling [71], | UML | + | + | C |
| Magic draw [67], Visual paradigm [108] | UML | + | + | C++, Java |

and timing diagram) is discussed below.

Approaches followed in [21] and [86] describe the applicability of UML timing diagrams for different domains at the design level. However, a model-based technique for visualizing the embedded software behavior in real time, using UML interaction diagrams is missing in these approaches. An architecture for gaining insight into executable models during runtime is proposed in [31]. In this work, an extension to the MDD approach provides an architecture for debugging models that execute on target systems. An example of debugging target behavior using state diagrams is described. However, an example of reconstructing the target behavior using UML sequence and timing diagrams is missing in this work.

From Table 2.2 it is evident that there is extensive tool support for specification of embedded software requirements using models and automatic code generation to obtain the system code. However, other than a few simulation approaches, only a few tools support finding errors in embedded software at the graphical modeling level [31]. For instance, industry standard MDD tools such as [11], [23] and [42] support modeling and design based on UML diagrams. These tools find wide applicability in industrial, research and development projects for embedded software development using a model-based approach. However, model-based debugging for embedded systems using such tools is limited in terms of their applicability for memory-constrained (small sized) embedded systems (which have limited RAM/ROM). For instance, the “live animation” features from [42] cannot be used for memory constrained real time embedded systems. This is because of the need for significant memory on the embedded system, protocol overhead such as TCP/IP over Ethernet and performance overhead. These techniques result in inefficient communication between the target and the host in the small embedded systems. Further, such methods introduce dynamic source code instrumentation, which can significantly influence the temporal behavior of the real time critical embedded systems. In addition, they do not support model-based debugging or visualizing the target behavior using the newly introduced UML timing diagrams.

### 2.2.2 MBT for embedded systems

This thesis proposes an integrated model-based approach and test framework involving both the MDD and the MBT phases for embedded systems (challenges (1) and (2)). In this context, the applicability of MBT for embedded systems is explored in this section by discussing the related research work and existing tool support for the MBT phase in embedded systems.

In general, the application of MBT methodology in several domains and application areas is discussed in [3], [10], [15], [25] and surveys/reviews such as [9], [19], [77]. For example, a subset of UML elements for MBT is proposed in [10]. This subset allows formal behavior models of the SUT to be designed, which can be manually interpreted to generate test suites. The applicability of model-based on the fly testing in the context of web applications using the *NModel* toolkit is described in [25]. This work deals with the MBT toolkit for a non-trivial case study. However, the case study is based on a web application and the toolkit is suitable for frameworks using the programming language C#. Similarly, there are several MBT techniques for testing desktop-based systems. But, the reusability of such techniques for their applicability in the embedded systems domain needs a detailed study.

The MBT methodology proposed in [3] models properties of the environment, its interactions with embedded systems and potentially unsafe situations triggered by failures of the embedded SUT. The proposed methodology uses standards such as UML, MARTE and OCL. The models describe both the structural and the behavioral properties of the embedded systems. However, this work focuses on using the developed models in an embedded system environment simulator in the programming language Java and lacks extensive experimental evaluation. MBT for embedded systems in the automotive domain is carried out using Matlab, Simulink and Stateflow tool chain in [122]. The automotive specific MDD is taken into account to illustrate the problems and prepare the background for further considerations on Quality Assurance (QA). However, a design model based on UML or a specification of test artifacts based on UML/UTP is missing. The remainder of this section discusses the related work pertaining to MBT, based on the classification of the MBT process.

Based on the classification of the steps involved in the MBT process (Figure 2.7), various MBT methodologies have been reported for different application domains [9], [15]. A few approaches discussed in the literature concentrate on steps (a), (c) and (e) in Figure 2.7. These steps pertain to specifying/generating the test model, concretizing the tests and analyzing the test results [9]. On the other hand, a vast number of MBT approaches discussed in the literature concentrate on generating the test cases using models [10], [19], [38], [77], i.e., step (b) in Figure 2.7. However, an infrastructure to generate the test input data, from models such as state diagrams is missing. Note that this aspect refers to an automated and systematic procedure to generate the test input data to be used in the test cases. This does not refer to the test suites/test case generation from models. For example, infrastructure support to generate the test cases from state charts is discussed in [39], [53], [57], [76]. Similarly, test case generation from finite state machines is discussed in [40]. These concentrate on generating the test cases from models. However, a procedure for the systematic derivation of test input data from state diagrams (specified for example in system design model/abstract test model) is not dealt with. Similarly, infrastructure support for conveying the test data to the SUT is not explored.

While there is extensive tool support for the MDD phase [23], [42] (e.g. using UML), [68] (e.g. using matlab/simulink) there is only limited tool support for executing the model-based test cases

in the embedded system [43] (e.g. using UML). Some commercial tools [64], [113] provide support for various steps in MBT. However, the methodology involved in executing the test cases obtained from MBT for embedded systems is not discussed [19], [77]. Further, an infrastructure to handle the test data generated in the test models and executing model-based testing (i.e., Figure 2.7 step (d)) especially in resource constrained embedded systems is missing in the existing tools and approaches. In summary, based on the above discussion and the challenges outlined in this thesis, the two main gaps are,

- Lack of methodologies for automatic generation of test artifacts, i.e., the test framework, for executing the model-based test cases in the embedded system, i.e., step (d) in Figure 2.7
- Dearth of infrastructure support for automatic generation of test input data and conveying the test input data from the test framework to the embedded system (e.g. derivation from state diagrams in system design model/abstract test model), i.e., gaps in step (b) in Figure 2.7

This thesis focusses on these gaps and proposes a methodology for generating a test framework during the MBT phase, based on the system design model available in the MDD phase (challenge (2)). On the other hand, pertaining to challenge (1) mentioned in the introduction, this thesis aims to propose an integrated model-based approach and test framework for executing the model-based test cases in the embedded system. While doing so, the main challenge is to employ corresponding modeling languages for both the MDD and MBT phases. In this context, UML is used for the MDD phase in this thesis. As the leading standard in model-based software engineering, UML is also being applied in the area of testing. With the UTP, additional model-elements have been defined enabling the specification of test goals, test procedures and test assessments for system components as well as for complete systems. Hence, a corresponding modeling language for the MBT phase which also provides infrastructure support for executing the model-based test cases, namely UTP, is chosen in this thesis.

### 2.2.3 MBT and UTP

This thesis aims to propose an integrated model-based approach and test framework, wherein corresponding modeling languages are envisioned to be used for the MDD and the MBT phases (challenge (1)). In this context, the usage of UML for the MDD phase and UTP for the MBT phase are explored in this thesis. Therefore, related work pertaining to the usage of UTP during the MBT phase is discussed in this section.

Related research work pertaining to the usage of UTP in MBT is available in [58], [61], [73], [98]. Mapping of concepts from MBT to UTP is discussed in [58]. An example from a commercial tool (using UML) is illustrated to show how the UTP can be used to incorporate MBT in the software engineering process. However, the actual test execution, implementation level details of the SUT and the test cases are not explained.

An approach to manually select and automatically transform the relevant parts of the design model into a basic test model that can be used for test generation is proposed in [73]. Though the test model is closely based on the UTP, the applicability of the discussed approach for embedded systems or empirical studies is missing. Similarly, an approach for MBT which is completely based on standards, which constitutes an essential factor for its usage in industrial software tools, is discussed in [61]. The inputs of the process are models described in UML 2.0, whereas the outputs consist of artifacts

of the UTP. The transformation process is based on QVT (Query-View-Transformation) scripts, a transformation language also adopted by the OMG.

A comparison of elements in UTP and TTCN-3 and their usage for various development and test processes is discussed in [98]. Similarly, research on the mapping of UTP to TTCN-3 is described in [7] and [121]. Potential applications of UTP in the context of software development process for communication standard, is discussed in [98]. It concludes that, UTP enables high-level test design along UML based design processes and combining both TTCN-3 and UTP together in an UML based development process makes UML applicable for software testing. Similarly, an experiment with a combination of UTP in conjunction with proprietary domain specific languages (DSLs) is discussed in [97]. It concludes that UTP supports model-based development of test suites in a very intuitive way, thereby enabling UML-based test modeling for application scenarios.

The aforementioned related work concentrates on using UTP, either in isolation or in combination with other languages, for MBT. However, an elaborate discussion on the application of the proposed approach and an empirical evaluation in practical example scenarios is not available.

#### MBT and UTP for embedded systems

Based on the related work discussed above, it is evident that unlike the extensive research for using UML in the MDD phase, especially for embedded systems, the applicability of UTP and its usage for embedded systems is still an emerging field. A significant reasoning for this could be based on the role of UTP. For instance, the UTP is used only to specify explicitly a typical architecture that can be associated with MBT, while the MBT framework automates the testing process. This implies that UTP as such does not specify how to carry out the testing process [58]. Hence, there are several MBT approaches for automating the testing process. Only a very few of them deal with the applicability and specification of a test framework which automates the test case execution process using UTP. This thesis concentrates on the generation of an integrated test framework based on UML and UTP at the host computer/design level. This test framework is envisaged to automate, as far as possible, the test case execution process on the host computer and only inserts the test input data to the embedded system.

The tool support for using UTP in MBT, especially for embedded systems is very limited. An add-on Rhapsody-Test Conductor [43] for a MDD tool [42] is the only available MBT tool which supports development of UTP-based test infrastructure, especially for embedded systems. This tool supports specifying the test artifacts using UML and UTP notations and subsequently executing test cases in the embedded system. However, the methodology used by the tool to execute the resulting model-based test cases in embedded systems, involves significant source code instrumentation and intrusive testing technique. This approach is not suitable for memory-size constrained embedded systems. This can also result in affecting the real time characteristics of the embedded system during/after the test case execution process.

Surveys on MBT approaches [19], [77] identify the advantages and pitfalls in the existing MBT techniques. Based on the conclusions in [19], it is evident that the MBT approaches are usually not integrated with the software development process, for instance the MDD phase. It is also apparent that there is a need for integration tools which combine both the MDD and MBT phases for software engineering projects. It is intuitive to perceive that while transferring an integrated (MDD and MBT) model-based approach to real-life software engineering projects, it is beneficial to minimize the use

of tools and modeling languages for the individual phases of MDD and MBT. Nevertheless, such comprehensive approaches are currently in nascent stages in both industry and academia [19]. The MBT approach and tool support presented in [30] is suitable for executing model-based test cases, with minimal overhead, in embedded systems. However, it does not deal with an integrated model based approach, i.e., combining MDD (using UML) and MBT (using UML and UTP) for embedded systems. On the other hand, tools such as [42] (e.g. using UML for MDD) and [43] (e.g. using UTP for MBT) together could provide integrated model-based approach for embedded system scenarios. But, the testing methodology is intrusive and not applicable for resource constrained embedded systems.

Thus, relating to challenges (1) and (2) introduced in this thesis and the aforementioned discussion on related work, the following can be summarized. An integrated model-based approach and test framework using UML for MDD and UTP for MBT, especially in the context of their applicability in resource constrained embedded systems is missing in the existing approaches. On the other hand, there are several approaches in the literature which deal with model-based methodologies and MBT for various application purposes. Most of these approaches concentrate on generating the test cases from the given test model. However, none of these approaches deal with generating a test framework which includes test artifacts for executing these model-based test cases in memory-size constrained and code-size critical embedded systems. Also, infrastructure support to handle the automatically generated test data and conveying the test input data to the embedded system is missing. A majority of the approaches use non-standard modeling languages, except [3] which uses UML and MARTE for embedded system environment modeling. Moreover, the approaches in the literature do not combine test framework generation with UTP for embedded systems and executing MBT with such a test framework in embedded systems. Last but not least, none of the related work assesses their MBT methodology on an actual real-world (memory-size and code-size critical) embedded system case study and applicability of MBT in such scenarios.

To address these gaps, an integrated model-based approach and test framework is proposed in this thesis, relating to challenges (1) and (2). The proposed approach generates the necessary test artifacts (i.e., the test framework) for executing the model-based test cases (i.e., step (d) in Figure 2.7) in embedded systems. The proposed approach can be considered as a generic approach for various application domains and modeling languages. In the prototype developed in this thesis, the test framework is generated, using UML and UTP concepts at the host computer/design level by a test framework generation algorithm (proposed in this thesis).

### 2.2.4 Runtime monitoring for embedded systems

The test framework proposed in this thesis, is envisioned to make use of a runtime monitoring methodology in the embedded system to insert the test input data/test stimuli in the embedded system. This relates to challenge (3) introduced in this thesis. Related work pertaining to this aspect is discussed in this section.

Software monitoring has been in use for over 35 years for a variety of domains and application purposes [41], [78], [85], [104] and is classified in [16], [84]. Some of the domains in which runtime monitoring is applied include, distributed systems [35], [107], fault-tolerant systems, real time critical systems [81], [109] and embedded systems [13], [100], [116]. Classification of monitoring approaches based on hardware/software/hybrid probes used is available in early surveys such as [78], [91]. The

categorization continues to be used in recent work such as in [103]. The classifications of runtime monitoring approaches can be used for a variety of application purposes in embedded systems. This depends on the applicability of the monitoring approach (e.g. its pros and cons) for the end-user's requirements. Potential applications of the runtime monitoring methodology for embedded systems include profiling, performance analysis, software optimization, software fault-detection, diagnosis and recovery, debugging, verification and testing. Relating to challenge (3) introduced in chapter 1, the following concentrates on the related work for the application of runtime monitoring in the context of executing model-based test cases in the embedded system. In addition, related work pertaining to non-intrusive monitoring methodologies is outlined.

Research on various runtime monitoring approaches described in the literature have been classified in works such as [16] and [84]. The applications of runtime monitoring approaches often employ a target output/computer to diagnose and interpret the results in formats such as plain text and graphs. On the other hand, in the recent decade, with the advent of model-driven methodology [79] the applicability and usage of models such as UML diagrams are considered not only for software development but also for model-based runtime monitoring and testing of software systems [49]. In case of embedded systems, this methodology is a significant step forward in terms of visualizing, debugging and testing the embedded system behavior in real time using high level models, such as UML diagrams in real time (at the host computer/design level). In this thesis, such a runtime monitoring methodology is employed for sending and receiving the test data (e.g. test input, test results) between the host computer (test framework) and the embedded system. This is achieved by communicating the test data via a debug communication interface between the host computer and the embedded system, collectively termed as *back annotation*.

A study involving back-annotation of timing information into a formal hardware model is conducted in [117]. A tool for interactive static worst case execution time analysis with continuous feedback to the developer in the form of back annotations is provided in [37]. The study is conducted on Java microprocessors which are not necessarily the preferred choice in embedded system with 8 or 16 bit microcontrollers. In [51], an implementation of an event-driven hardware/software collaborative monitor system, enabling system-level monitoring on the target at different abstraction levels is presented. The described monitor system is claimed to collaborate seamlessly with other components in a model driven testing tool chain. The objective of the presented monitor system is to provide an appropriate means for runtime behavior capture and analysis of the SUT. However, this technique can result in dynamic source code instrumentation, which could affect the real time characteristics of the embedded system.

Related research work on non-intrusive monitoring methodologies is available in [27], [32], [36]. The importance of non-intrusive monitors and the problems with intrusive software monitoring systems are discussed in [36], where a less intrusive software monitoring system is presented. A study on non-intrusive and 'low' intrusive monitoring in the context of more complex systems, in particular embedded systems, is carried out in [27]. Non-intrusive black and white box testing of embedded software against UML models is described in [32]. In this work [32], an approach for specifying and executing tests on UML models is described. While the test cases are generated from the test model, they are evaluated against the real system response of the embedded software. The technical basis for collecting the required information is a commercial debugging hardware-device for embedded processors capable of real time tracing. Similarly, a hardware-based monitoring methodology is used

for MBT in the *eMote* project [30]. The techniques outlined in [30], [32] aim at minimizing the instrumentation overhead for collecting the trace data, but do not eliminate it completely. Further, the proposed methodology does not make use of an integrated model-based approach or corresponding modeling languages for the MDD (e.g. UML) and MBT (e.g. UML and UTP) phases. In line with challenges (1), (2) and in particular (3) and also based on the above discussion on related work pertaining to runtime monitoring of embedded systems, the gaps in the existing approaches can be summarized as follows.

- Dearth of minimally intrusive monitoring methodology, with bounded measurable overhead (time, memory) parameters, for inserting and executing the test stimuli in resource constrained embedded systems.
- Lack of applicability of a generic, software-based, runtime monitoring methodology in an integrated model-based approach and test framework (which employs UML and UTP) for executing the model-based test cases in the embedded system.

This thesis focusses on these gaps and proposes a generic, software-based, runtime monitoring methodology for resource constrained embedded systems (challenge (3)). Such a runtime monitoring methodology, with bounded and measurable overhead, is primarily used by an integrated model-based approach and test framework towards executing the model-based test cases in the embedded system (refers to challenges (1) and (2)). In addition, the runtime monitoring methodology is also used for visualizing the target behavior in real time, at the host computer, using UML interaction diagrams such as sequence and timing diagrams (e.g. [50], [49]).

### 2.2.5 Industrial evaluation scenarios

Relating to challenge (4), this section provides a brief outline on the related work pertaining to the evaluation scenarios of the MDD/MBT approach. A systematic review and survey on MBT approaches available in [19] concludes that most MBT approaches are not evaluated empirically and/or not transferred to the industrial environment. Related research work on MBT, in several cases, deals with application scenarios pertaining to desktop-based systems or web application development [9], [19]. While there are examples dealing with embedded system scenarios [32], an experimental evaluation on a real-life case study is not available. For instance, the development of a central spark extinguishing system for a real-life industrial example project is described in [18]. However, the MBT approach, for this scenario is not described. Contributions to the UTP specification in [6] and [123] deal with the application of the UML and UTP for MDD and MBT phases for the development and testing of a roaming algorithm for bluetooth devices. Moreover, studies based on an integrated model-based approach involving corresponding modeling languages for real-life examples are not carried out. While [30] deals with an embedded software application example and executing model-based testing in embedded systems, it does not involve UTP. Moreover, it uses a proprietary In-Circuit Emulator (ICE) for executing the model-based test cases in the application scenario. This can be classified as a hybrid, on-chip monitoring mechanism and does not qualify for a categorization as a generic monitoring methodology.

In this thesis, a prototype implementation of the proposed approach, relating to challenges (1), (2) and (3), is envisaged. The prototype of the test framework and its components are evaluated in

a real-life embedded software engineering example comprising of a MIDI system analyzer case study. This addresses challenge (4) outlined in the introduction. In addition, an example of a industrial case study evaluated using the approach proposed in this thesis is outlined coarsely in this document.

# Chapter 3

# Integrated Model-Based Approach and Test Framework: An Overview

This chapter provides a general overview on the proposed integrated model-based approach and test framework for embedded systems. The MDD and MBT phases in the proposed approach are described in section 3.1 and 3.2 respectively. By this, the goals pertaining to the question of how to develop an integrated model-based approach and reuse models in the MBT phase from the MDD phase are addressed (challenge (1)). Corresponding modeling languages are envisaged to be used for both the MDD and MBT phases. This discussion results in shaping the components of the generic test framework proposed in this thesis (challenge (2)). The key idea of the proposed test framework is to enable test automation (and test case execution) at the host computer, while inserting only the test input data to the embedded system. To realize this, the requirements of a generic software-based runtime monitoring methodology and its applicability in the proposed test framework approach are discussed in section 3.3 (challenge (3)).

The integrated model-based approach and test framework for executing the model-based test cases in embedded systems, proposed in this thesis, is shown in Figure 3.1. The two phases for the embedded software development and testing in this approach are the MDD (Figure 3.1 - left side) and the MBT phases (Figure 3.1 - right side) respectively. The activities in these two phases, in the proposed approach, are described in section 3.1 and 3.2 respectively.

## 3.1 Model Driven Development phase

Relating to challenge (1), which deals with the applicability of an integrated model-based approach and usage of corresponding modeling languages for MDD and MBT phases, a background on MDD and MBT is provided in section 2.1 (chapter 2). Given this background and respective related work discussed in section 2.2, in this section an outline of the steps involved in the MDD phase, in the proposed integrated model-based approach, is derived.

The series of steps involved in MDD, discussed in section 2.1, are applied for arriving at the steps involved during the MDD phase in the proposed approach. As seen in Figure 3.1 (on the left), during the MDD phase based on the requirements specification, the design model for the application software

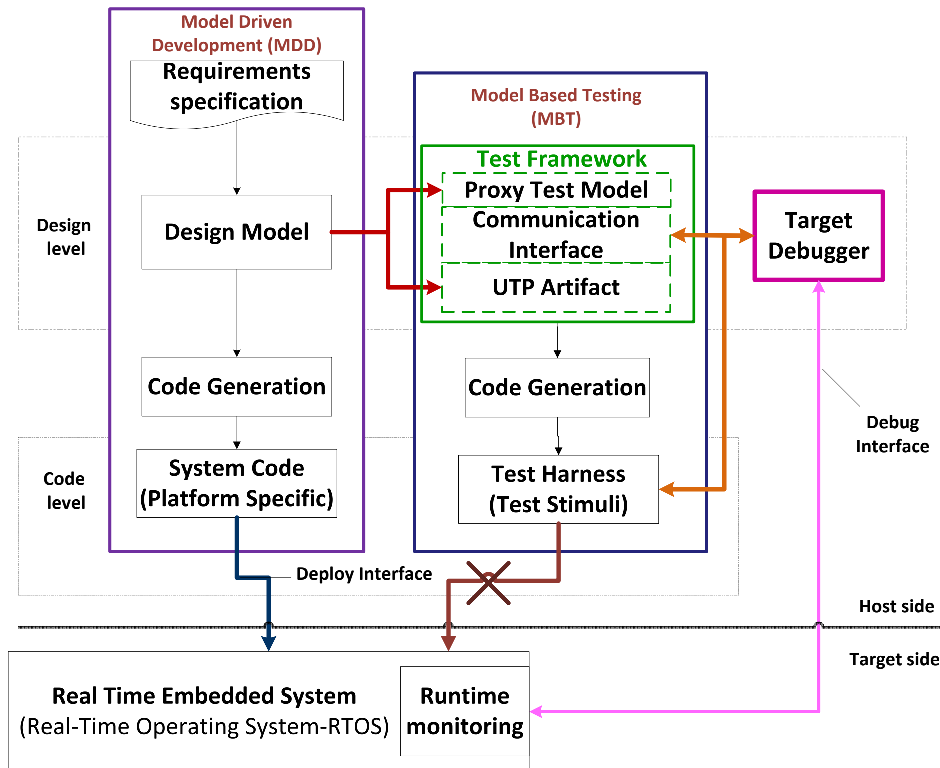



Figure 3.1: Integrated model-based approach and test framework for embedded systems

to be developed for an embedded system is specified using a modeling language. Several modeling languages and tools exist for modeling embedded software systems. Some of the widely used modeling languages are UML, SysML [99] and tools such as Rhapsody [42], Matlab [68], LabVIEW [60]. In the next step (Figure 3.1), based on the design model, the platform specific system code is obtained using an automatic code generation process. The system code can be generated in programming languages such as "C" and "C++" based on the end-user's requirements and the embedded system under consideration. The aforementioned steps for the MDD phase in the proposed approach are in line with the generic steps outlined in section 2.1.

The system code thus obtained runs on the embedded system which comprises of an underlying Real-Time Operating System (RTOS). VxWorks [114], Windows CE [119] and embedded linux [65] are some examples of the RTOS frameworks that find wide applicability in embedded software projects in the recent years. Some open source projects such as "The FreeRTOS project" [105] also provide RTOS support for embedded systems.

Based on the background and related work discussed in chapter 2, pertaining to the modeling language for the MDD phase (in line with challenge (1)), it can be stated that UML is a widely used general-purpose modeling language and a leading standard in model-based software engineering. Moreover, among the choice of the modeling language for embedded software engineering, UML can be considered as the most widely used. In this context, UML is chosen as the modeling language for specifying the design model. The main functionality of the application can be specified, for example using UML class diagrams. The detailed functionality and reactive behavior of each class can be

represented as a state diagram, for example using UML state charts.

Note that in the integrated model-based approach and test framework proposed in this thesis (Figure 3.1), the already established parts of the MDD phase are part of the proposed approach. On the other hand, during the MBT phase a test framework is generated based on input from the MDD phase, for executing the model-based test cases in the embedded system.

## 3.2 Model Based Testing phase

The challenge (1) introduced in this thesis deals with the applicability of an integrated model-based approach and the usage of corresponding modeling languages for test automation in embedded systems. Addressing the aforementioned challenge, an integrated model-based approach (comprising of the MDD and MBT phase) is proposed in this thesis (Figure 3.1). The steps involved in the MDD phase and the choice of the modeling language for the MDD phase is outlined in the previous section.

In this section, the activities involved in the MBT phase in the proposed test framework are outlined. The choice of the modeling language for the MBT phase, thereby enabling the use of corresponding modeling languages for the MDD and MBT phases is discussed. This discussion aids in shaping the components of the generic test framework proposed in this thesis, thereby relating to challenges (1) and (2).

The system code generated during the MDD phase needs to undergo a thorough quality assurance process. The disadvantages in the existing MBT methodologies towards executing the model-based test cases in embedded systems, relating to challenges (1) and (2) have been outlined in chapter 2. In summary, based on the background and related work pertaining to the applicability of an integrated model-based approach and test automation towards executing the model-based test cases in embedded systems, the main gaps this thesis aims to address are the following:

- Applicability of an integrated model-based approach and usage of corresponding modeling languages for the MDD and MBT phases (challenge (1)). An example usage of corresponding modeling languages for the MDD and MBT phases is the usage of UML for MDD and UTP for MBT phases.
- Automatic generation of test artifacts, i.e., the test framework, for executing the model-based test cases in the embedded system. This refers to step (d) in Figure 2.7.
- Infrastructure support for automatic generation of test input data and conveying the test input data from the test framework to the embedded system (e.g. derivation from state diagrams in the system design model/abstract test model), i.e., gaps in step (b) in Figure 2.7.
- Executing the model-based test cases in the embedded system with minimal overhead/non-intrusive monitoring methodology. This pertains to challenges (2) and (3) which deal with the automatic generation of the test framework and executing the model-based test cases without introducing significant overhead in the embedded system.

Note that this thesis does not deal with the generation of test cases from the test models (e.g. [9], [43]). Instead, this thesis concentrates on an integrated model-based approach and test framework which enables test automation for executing the model-based test cases in resource constrained embedded systems.

Therefore, to achieve a significant gain in employing a MDD and MBT methodology, a reliable technique for executing the model-based test cases (during the MBT phase) should be used for quality assurance aspects. In this context, this thesis proposes an integrated model-based approach and test framework for executing the model-based test cases in the embedded system (challenges (1) and (2)). Hence, given the system design model and the chosen SUT, a test framework is generated automatically at the host computer. The test framework is used to aid the test case execution process without downloading the test harness on the embedded system (which is the case in existing tools and techniques-indicated with a cross mark in Figure 3.1). The test framework generated automatically during the MBT phase, is used primarily for two purposes namely,

1. Carry out the test case execution process at the host computer, while only inserting the test data input to the embedded system with the aid of a runtime monitoring mechanism.

2. Visualize the test results at the host computer as models, for example using UML sequence diagrams, based on the trace data obtained from the embedded system.

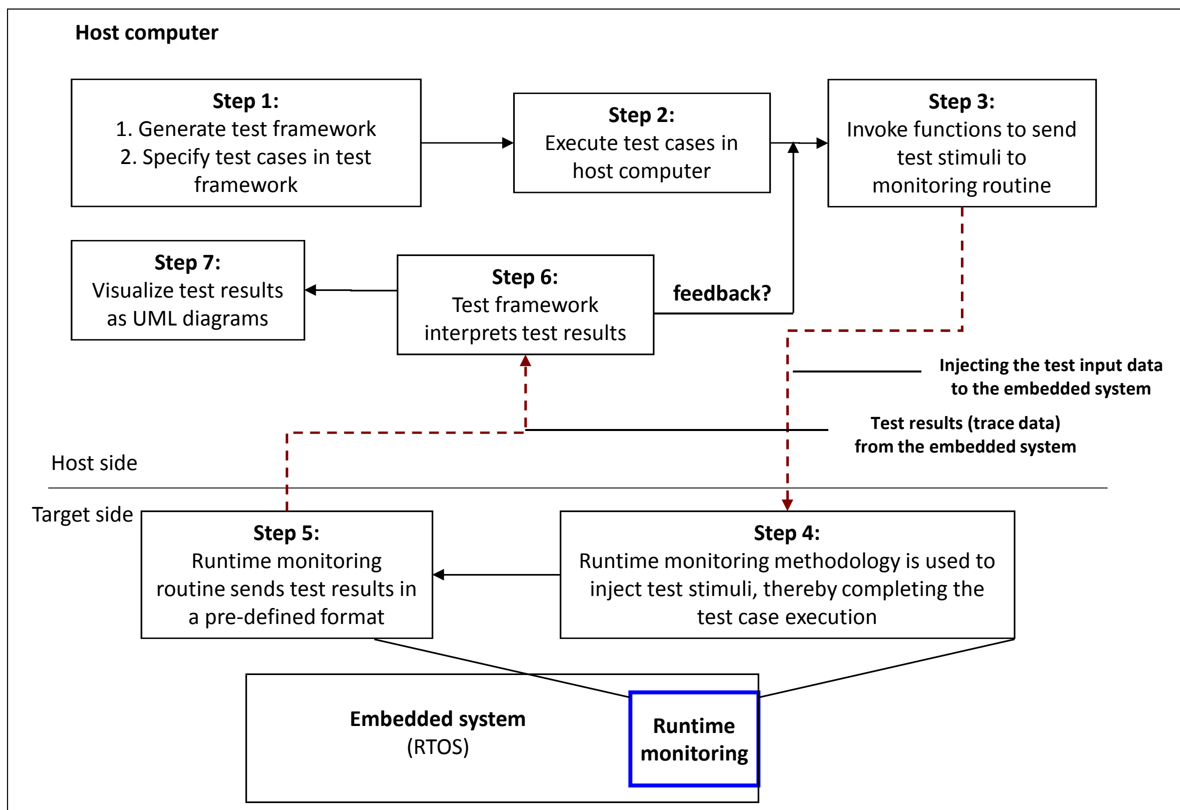



Figure 3.2: Steps involved in executing model-based test cases in the proposed approach

To gain further understanding on the applicability of the test framework towards executing the model-based test cases, consider a series of steps illustrated in Figure 3.2. This series of steps provides an overview of the envisioned usage of the test framework for executing the model-based test cases using the proposed approach. Note that in Figure 3.2, the dotted lines represent the communication between the host computer and the embedded system. In other words, these lines represent injecting the test

input data from the host computer to the embedded system and receiving the test results (trace data) from the embedded system by the host computer.

In the first step, given a chosen SUT and the system design model, a test framework is generated at the host computer. The model-based test cases are specified manually in the automatically generated test framework. In the second step, the test cases are executed at the host computer, with the aid of the test framework. In step 3, the respective functions to send the test input data from the host computer to the embedded system are invoked. In step 4, the test input data is executed in the target by the runtime monitoring routine in the embedded system. Once the test input data is executed at the target, the runtime monitoring routine (in the target) sends the test result to the host computer (step 5). The test framework interprets the test results in the next step, i.e., step 6. If there is any further action to be taken based on the test results, then a feedback is provided to step 3, to send the respective test stimuli (based on the feedback) to the target. Once the test results are interpreted, they are visualized as UML diagrams (e.g. UML sequence diagrams) in the next step, i.e., step 7. This series of steps illustrated in Figure 3.2 indicate that the test case execution process is carried out at the host computer. Then, only the test input data is injected to the embedded system, and executed at the target with the aid of a runtime monitoring methodology at the target. This approach aims to address the goal pertaining to minimizing the overhead involved for executing the test cases in the embedded system (challenges (2) and (3)).

**Additional components required for the test framework**

Based on the above series of steps, it is clear that in order to support the MBT phase (and the test framework) for executing the model-based test cases in the embedded system, additional components are necessitated.

- For instance, a component is required in the embedded system to inject the test stimuli (received from the host) and send the trace data about the test results (to the host).
- Similarly, an additional component (other than the test framework) is required to decode the test input (from the test framework) and interpret the test results (from the target) before it can be used by the target and the test framework respectively.

These components are required to support the test case execution process at the host computer and insert the test input data alone to the embedded system (described in the aforementioned series of steps). By this way, the test harness is not downloaded on the target for executing the model-based test cases in the embedded system (Figure 3.1).

These two additional components which aid the test framework for executing the model-based test cases are the *target debugger* on the host computer and a monitoring component based on the *runtime monitoring* methodology in the embedded system, as seen in Figure 3.1. A detailed description leading to the choice of a runtime monitoring mechanism and the applicability of the target debugger in the context of the proposed test framework are discussed in section 3.3. A brief description on the envisaged functionality of these components is provided below, to place in context the components of the test framework which are described in section 3.2.1.

A target debugger at the host, is used for two purposes. First, it is used by the test framework to inject the test stimuli (e.g. test input data) to the embedded system via the runtime monitoring

methodology (Figure 3.1). Second, it is used to convey the test data (e.g. test results) received from the runtime monitoring mechanism (in the embedded system) to the test framework (Figure 3.1).

### 3.2.1 Components of the test framework

In the proposed approach, during the MBT phase (Figure 3.1), given the design model and the chosen SUT, a test framework generation algorithm (proposed in this thesis) generates the necessary artifacts (i.e., the test framework) for executing the model-based test cases in the embedded system. Corresponding modeling languages are envisaged to be used for the MDD and the MBT phases in the proposed approach, in line with challenges (1) and (2). As discussed in chapter 2, a concrete modeling language/standard for the specification of infrastructure support for executing MBT in the SUT, is not available until the introduction of the UTP. Moreover UML is chosen as the modeling language for the MDD phase, in the proposed approach. Hence this thesis aims to explore the applicability of UTP as a corresponding modeling language for specification of test infrastructure support for executing the model-based test cases in the embedded system during MBT (challenges (1) and (2)). Therefore, relating to these goals, UML is chosen as the modeling language for the MDD phase and UML, UTP are chosen as the modeling languages for the MBT phase in the proposed approach. Whereas the components of the test framework are specified in UML and UTP during the MBT phase, the test framework is generated automatically by a test framework generation algorithm (based on the chosen SUT and design model) proposed in this thesis.

The elements of the test framework are grouped into three main components, based on their functionality. The three groups are (Figure 3.1)

- Proxy test model
- Communication interface and
- UTP artifact

The elements of the test framework are aggregated into the aforementioned groups, based on the following reasoning.

The test framework is used for two main purposes (Figure 3.2). This involves executing the test cases at the host computer (thereby inserting the test input data to the embedded system) and visualizing the test results as UML diagrams. Hence, UML-based artifacts are required to mirror the test case execution process at the host computer, based on the test results (trace data) obtained from the embedded system. This component shall comprise of a corresponding module (e.g. class) w.r.t each module (e.g. class) associated with the chosen SUT in the system design model. Thus, the *proxy test model* component is an abstract copy of the design model and does not inherit any functionality of the design model. It comprises of UML-based components such as classes, derived based on the given system design model and the chosen SUT. It also comprises of the functionality to store and convey the test input data, in a pre-defined format, from the test framework to the embedded system.

To convey the test data from the proxy test model to the embedded system via the target debugger, a *communication interface* component is necessitated. The main functionality of this component is to enable uninterrupted communication between the test framework and the target debugger, as seen in Figure 3.1. This addresses the goals pertaining to executing the model-based test cases in the embedded system and communicating the test results to the host computer (challenges (1), (2), (3)).

The *UTP artifact* component, as the name implies, is used to provide infrastructure support for the test case execution process at the host computer. Therefore to aid the test case execution process, the UTP based elements from the *test architecture* concept group is chosen for this component. This pertains to the goals regarding the usage of corresponding modeling languages for the MDD and MBT phases, for example UML for MDD and UML, UTP for MBT (challenge (1)). A detailed background on UTP and the elements from the test architecture concept group is provided in section 2.1. Thus, based on the functionality of the various elements in the test framework, its components are aggregated into three main groups as outlined above (challenges (1) and (2)). The following section describes the functionality of each of the aforementioned components in detail.

### 3.2.1.a Proxy test model

This component comprises of a corresponding test module with respect to each module associated (e.g. in the association end) with the chosen SUT in the system design model. For example, consider that three classes $a$, $b$ and $c$ are available in the system design model, such that $a \rightarrow b \rightarrow c$, i.e., $a$ is associated with $b$, $b$ is associated with $c$, as seen in Figure 3.3. But class $a$ is not associated with class $c$. Consider that class $a$ is the chosen SUT. Then, corresponding to this, the proxy test model

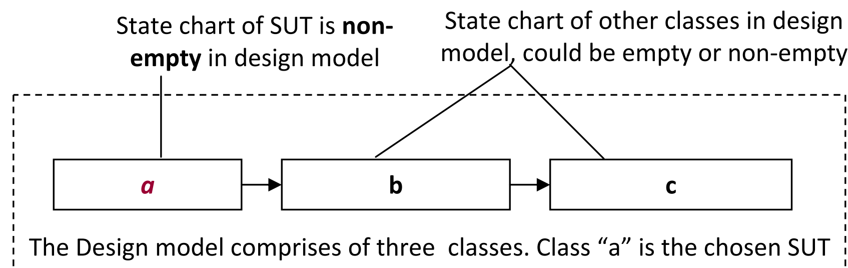



Figure 3.3: Example of a system design model and chosen SUT

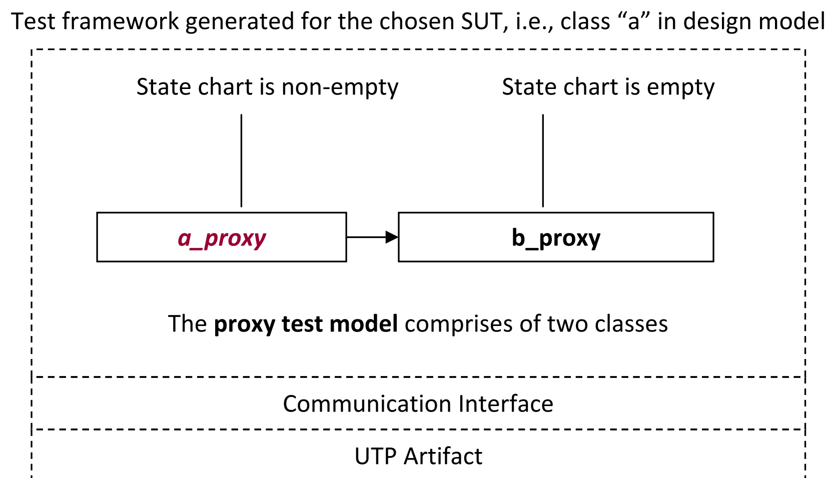



Figure 3.4: Example showing the automatically generated test framework (based on the design model shown in Figure 3.3) with proxy classes for the chosen SUT

component in the test framework comprises of classes *a_proxy* and *b_proxy* as seen in Figure 3.4. On the other hand, a class *c_proxy* corresponding to the class *c* in the design model is not created in the proxy test model. This is because, the chosen SUT i.e., class *a* is not associated with class *c*. All the operations and attributes from the classes *a* and *b* are copied to *a_proxy* and *b_proxy*. However, the operation/function body in the classes *a_proxy* and *b_proxy* are empty. Thus, the proxy test model (*a_proxy* → *b_proxy*) consists of only an abstract design model with no functionality inherited/derived from the system design model. With this, the proxy test model is used to mirror the test case execution process (at the embedded system), on the host computer based on the decoded test results from the target.

A state chart of the SUT is automatically generated in the proxy test model corresponding to the state chart of the SUT in the design model. The main functionality of this state chart is to (store and) convey the test input data corresponding to the transitions in the state chart of the SUT in the system design model (Figure 3.4). For example, if class *a* in the system design model is the chosen SUT, then the class *a_proxy* comprises of a state chart with the respective test input data for the chosen SUT (here class *a*). The transitions in this state chart, with their respective test input data, are invoked during the test case execution process, which is carried out by the test framework on the host computer. During this process, the aforementioned test input data, in a pre-defined format, is injected to the embedded system with the help of a monitoring routine in the embedded system. This addresses the gaps mentioned in the previous section regarding the automatic generation of test artifacts (i.e., the test framework) and infrastructure support for an automatic generation of test input data and conveying the test input data from the test framework to the embedded system. The state chart of the other classes, other than the chosen SUT, in the proxy test model is empty. For instance, the state chart of the class *b_proxy* is empty in the above example (Figure 3.4). This is because in the test framework approach outlined in this thesis, only the state chart of the SUT comprises of the test input data for the test cases specified, corresponding to the SUT, in the test framework.

**Test input data**

The test input data conveyed by the state chart of the SUT in the proxy test model, can be sent in a pre-defined frame format. Since this thesis deals with embedded software systems primarily modeled using an event-driven approach, a frame format for sending the test input data comprising of an event (and its source, destination, parameters) is proposed in this thesis. The proposed frame format for sending the test input data, comprising of an event and its related information (source, destination, event parameters), is shown in Figure 3.5. In this pre-defined frame format, each component in the

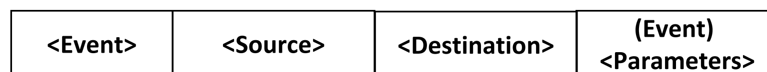



Figure 3.5: Frame format for test input data sent to the target debugger by the test framework

test data is separated by a delimiter. The test input data, in the pre-defined format, is conveyed to the target debugger with the help of functions in the communication interface component of the test framework (Figure 3.1).

For example, consider a test case which needs to inject an event *evAtoB()* from class *a* to class *b*,

i.e., $a \xrightarrow{evAtoB()} b$. The test data input for this example in the pre-defined format is *<evAtoB>* *<a>* *<b>*. Here, the *event*, *source*, *destination* in the test input data are *evAtoB*, *a* and *b* respectively. Note that there are no parameters for the event *evAtoB()* in this example.

Similarly, consider an example of a test case which needs to inject an event (with a parameter), *evToggle(int LedNr)* from a class *Controller* to a class *LED*, to turn on an LED (in the target) indicated by *LedNr*, i.e., $Controller \xrightarrow{evToggle(LedNr)} LED$. The parameters for the test input data, for this example, in the pre-defined format is *<evToggle>* *<Controller>* *<LED>* *<LedNr>*. Here, the *event*, *source*, *destination* and *event parameters* in the test input data are *evToggle*, *Controller*, *LED* and *LedNr* respectively.

**Test results**

The test results received from the embedded system via the target debugger are also in a pre-defined frame format as shown in Figure 3.6. Corresponding to the test input data, the test result comprises

| <Event> | <Source> | <Destination> | <Current Time> | <Current State> | (Event) <Parameters> |
|---|---|---|---|---|---|

Figure 3.6: Frame format for test results sent by the target debugger to the test framework

of the event that was executed in the embedded system and its corresponding source, destination and parameters. It also comprises of the current time at which this event was executed at the embedded system. The current state of the destination object, after this particular event was executed at the target is also available in the test results. Thus, all the data required to interpret the test result for a corresponding input data is available in a pre-defined frame format. This test data (i.e., the test result) is interpreted by the decoding functions in the test framework. Corresponding to the decoded data, the respective proxy classes in the proxy test model are invoked by the test framework. This action results in mirroring the test case execution process at the host computer, based on the results obtained from the actual test case execution at the embedded system.

For example, the parameters for the test result for the above example in the pre-defined frame format (Figure 3.6) is *<evAtoB>* *<a>* *<b>* *<CurrentTime>* *<ON>*. In this example, the *event*, *source*, *destination*, *current time* and *current state* are *evAtoB*, *a*, *b*, *CurrentTime* and *ON* respectively. In this example, there is no parameter for the event *evAtoB()*. In the test results, the *CurrentTime* value is represented in a time unit.

For the *evToggle(int LedNr)* example, the parameters for the test result are *<evToggle>* *<Controller>* *<LED>* *<CurrentTime>* *<ON>* *<LedNr>*. In this example, the *event*, *source*, *destination*, *current time*, *current state* and *event parameters* are *evToggle*, *Controller*, *LED*, *CurrentTime*, *ON* and *LedNr* respectively.

### 3.2.1.b Communication interface

This component is responsible for two functionalities, namely,

- It is used by the test framework to send and receive test data between the target debugger and the host computer and

- It encompasses the functionality to decode, interpret and invoke the respective functions based on the received message (e.g. test results) from the embedded system via the target debugger.

This component needs to run in its own thread of execution in order to enable uninterrupted communication between the test framework and the target debugger.

#### 3.2.1.c UTP Artifact

The UTP artifact component in the test framework provides structural/infrastructure support towards executing the model-based test cases in the embedded system. This pertains to addressing challenges (1) and (2), wherein usage of corresponding modeling languages for MDD and MBT phases and applicability of such an approach towards executing MBT in embedded systems is envisioned. As the name implies, this component is created based on the artifacts in the UTP. As described in chapter 2, the UTP profile introduces four concept groups namely, *test architecture*, *test behavior*, *test data* and *test time* [111], [123]. The entire set of UTP stereotypes for these concept groups is shown in Table 3.1. Among the four concept groups, the *test architecture* group provides a set of

Table 3.1: UTP concept groups and stereotypes (concise subset)

| ***Test Architecture*** | **Test Behavior** | **Test Data** | **Test Time** |
|---|---|---|---|
| SUT ∗ | Test objective | Data pool | Timer |
| Test component ∗ | Test case ‡ | Data partition | Time zone |
| Test context ∗ | Defaults | Data selector | |
| Test configuration ∗ | Validation action | Wild cards | |
| Test control ∗ | Verdicts | Coding rules | |
| Arbiter ∗ | | | |
| Scheduler ∗ | | | |

∗ and ‡ : Concise set of UTP elements used in the UTP artifact component
∗ : Automatically generated ‡ : Specified manually

concepts to specify the structural aspects of a test context covering the test components, the SUT and their configuration. This thesis proposes a test automation approach which deals with automatically generating the test framework infrastructure and the architectural support at the host computer for executing model-based test cases in embedded systems (i.e., relating to challenges (1) and (2)). Hence, a concise subset of the UTP components (among the extensive elements shown in Table 3.1) based on the test architecture concept group is chosen in this approach. The elements from the UTP which are chosen for the proposed UTP artifact component in the test framework are marked with a "∗" and "‡" symbol in Table 3.1. A background on the functionality of each of the elements in the *test architecture* component of the UTP has been provided in section 2.1.

An important point to note is that, this thesis deals with test automation for executing the model-based test cases in embedded systems. It does not deal with automatically generating the model-based test cases from a given test model. Hence, the test cases are specified manually (at the design level) in the automatically generated test framework. Then, the test case execution process is automated by the test framework and its components. However, the test cases specified manually are also grouped as part of the UTP artifact component. The elements from the *test architecture* concept group of the UTP are generated automatically in the proposed approach (marked with an "∗" symbol in Table

3.1), whereas the test cases (marked with an "‡" symbol in Table 3.1) are specified manually in the automatically generated test framework.

## 3.3 Runtime monitoring

To perceive a minimally intrusive monitoring mechanism, this section begins with an analysis on the pros and cons of runtime monitoring approaches (thereby relating to challenge (3)). Note that a runtime monitoring methodology which introduces minimal, deterministic instrumentation overhead (i.e., measurable beforehand) in the source code is also referred to as time and memory-aware runtime monitoring in this thesis. The requirements for a time and memory-aware monitoring approach which is capable of sufficiently observing the target behavior to support model-based test case execution are outlined (challenge (3)). The requirements are discussed in the context of a generic, software-based runtime monitoring methodology.

The proposed monitoring methodology aims at minimizing the monitoring overhead (i.e., time and memory) in the embedded system. In order to achieve this goal, the test framework makes use of two additional components, namely a target debugger on the host computer and a runtime monitoring methodology at the embedded system. The *target debugger* (on the host) is used to aid in communicating the test input data and test results between the test framework and the *runtime monitoring methodology* in the embedded system (Figure 3.1). Thus, the target debugger acts as an information marshalling agent between the test framework on the host computer and the runtime monitoring routine in the embedded system. By this methodology, the test harness is not downloaded on the target (as indicated with a cross mark in Figure 3.1) in the proposed approach.

### 3.3.1 Time and memory-aware runtime monitoring

In order to perceive a time and memory-aware runtime monitoring methodology, i.e., with minimal and bounded overhead in the embedded system (challenge (3)), the classifications of the runtime monitoring methodology (already discussed in chapter 2.1) can be recalled. Based on the pros and cons of the runtime monitoring mechanisms, namely, software, hardware, hybrid and on-chip monitoring it can be reiterated that the main factor influencing the usage of a monitoring mechanism, especially in the case of embedded systems, is the monitoring overhead. However, in order to provide a generic approach towards runtime monitoring, this thesis proposes to make use of a software-based runtime monitoring methodology. Since one of the major goals of this thesis is to minimize the monitoring overhead (challenge (3)), the generic software-based runtime monitoring mechanism is envisaged to be a time and memory-aware runtime monitoring methodology.

On the other hand, an often discouraging factor for the use of software monitoring is the possible side effects (overhead) from having the target system to execute additional software. Not only such instrumented code is often significant, but in practice removed after the monitoring process is completed. This implies that the software that is monitored is not the same as the one that is delivered as the end-product. For example, MDD tools such as [42] support "live animation" of the target behavior at the host computer. By this feature, the embedded system behavior is visualized at the host computer using UML state charts and sequence diagrams in such tools. However, in order to use this feature the embedded system needs to have sufficient memory to accommodate the instru-

mentation code required to enable this feature. Moreover, this methodology results in enormous trace data to be sent back and forth between the embedded system and the host computer. Therefore, this methodology cannot be applied to resource constrained embedded systems.

To overcome the aforementioned drawbacks for software monitoring and at the same time propose a time and memory-aware runtime monitoring methodology for executing the model-based test cases in the embedded system, the following aspects are taken into consideration in the proposed approach.

- Generic monitoring routine which is independent of the test cases (size, complexity) to be executed on the target.
- Minimally intrusive runtime monitoring routine (e.g. a few bytes of memory).
- Modular software monitoring approach independent of the debug communication interface used.
- Minimizing communication overhead between the monitoring routine and the application (e.g. test framework) which is decoding and interpreting the trace data (i.e., the test results) at the host computer.

Since the test framework is envisaged to automate the test case execution process at the host computer, only the test input data is sent to the software-based runtime monitoring routine in the embedded system in a pre-defined format (Figure 3.1, 3.5). Hence, by this methodology, only the test input data is injected to the target and the corresponding test results are obtained as trace data from the target. This implies that the only instrumentation overhead required for executing the test cases in the embedded system is the software-based runtime monitoring overhead in the embedded system. However, with a minimal, generic monitoring routine and bounded, measurable/pre-determined overhead (memory, time), the software-based runtime monitoring routine can be accommodated even in the final production code. Now, the additional monitoring overhead (memory, time) which is known beforehand, can be accommodated during the earlier phases of the development cycle. This can be achieved by allocating additional resources (memory) or adjusting the scheduling properties (time). Hence, the monitoring mechanism used for executing the model-based test cases in the embedded system can also be termed as a time and memory-aware monitoring methodology. In this thesis, this generic, software-based runtime monitoring methodology is denominated as *target monitor*.

### 3.3.2 Target debugger

To support a runtime monitoring mechanism which is envisaged to be a generic routine on the embedded system, prior knowledge about the embedded software executed on the target must be explicitly available at the host computer. For example, when the test case execution process is carried out at the host computer and only the test input data is sent to the target (as in the case of the proposed approach), prior information regarding the components of the test input data must be made available beforehand at the host computer. To gain further understanding, let us consider an example of a test data input sent in a pre-defined format (Figure 3.5) from the test framework. The steps involved in sending the test input data to the embedded system without and with the target debugger are shown in Figure 3.7 and 3.8 respectively.

Consider an example (already introduced in section 3.2) of a test case which needs to inject an event *evAtoB()* from class $a$ to class $b$, i.e., $a \xrightarrow{evAtoB()} b$. Then, the test data input for this example

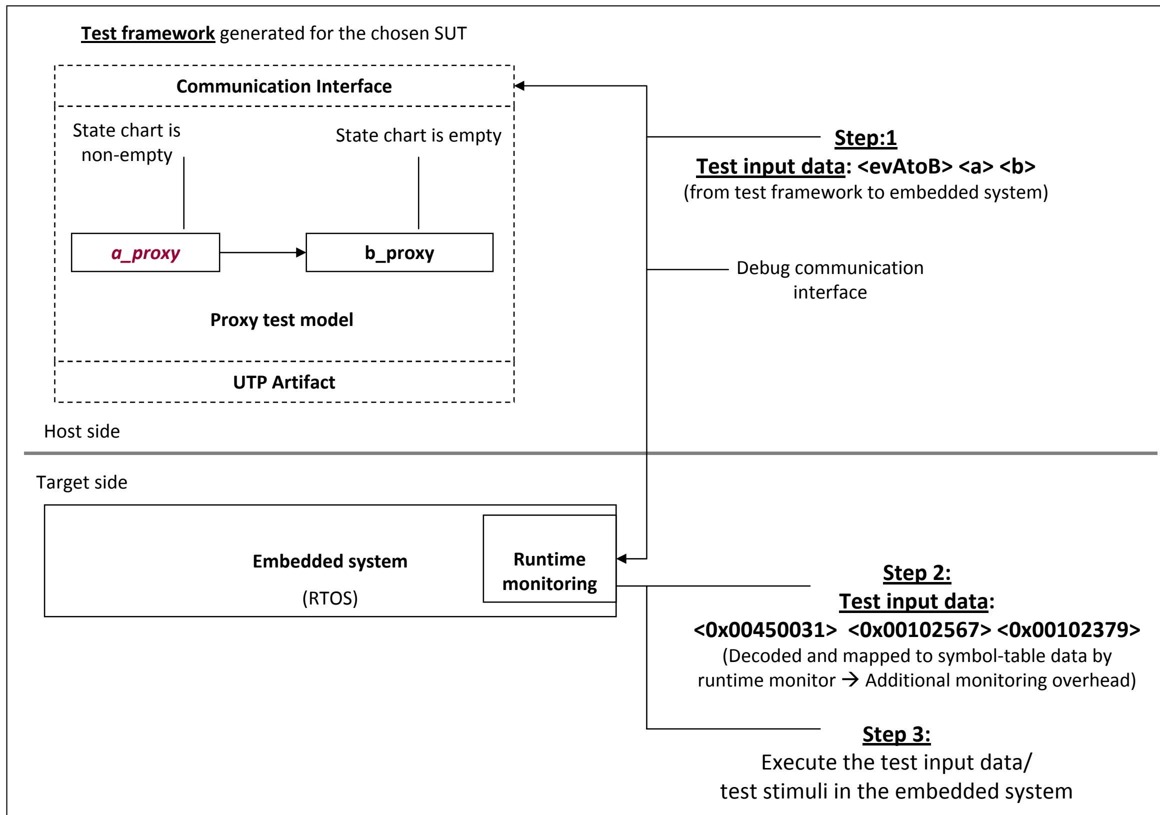



Figure 3.7: Steps involved in decoding the test input data without the target debugger

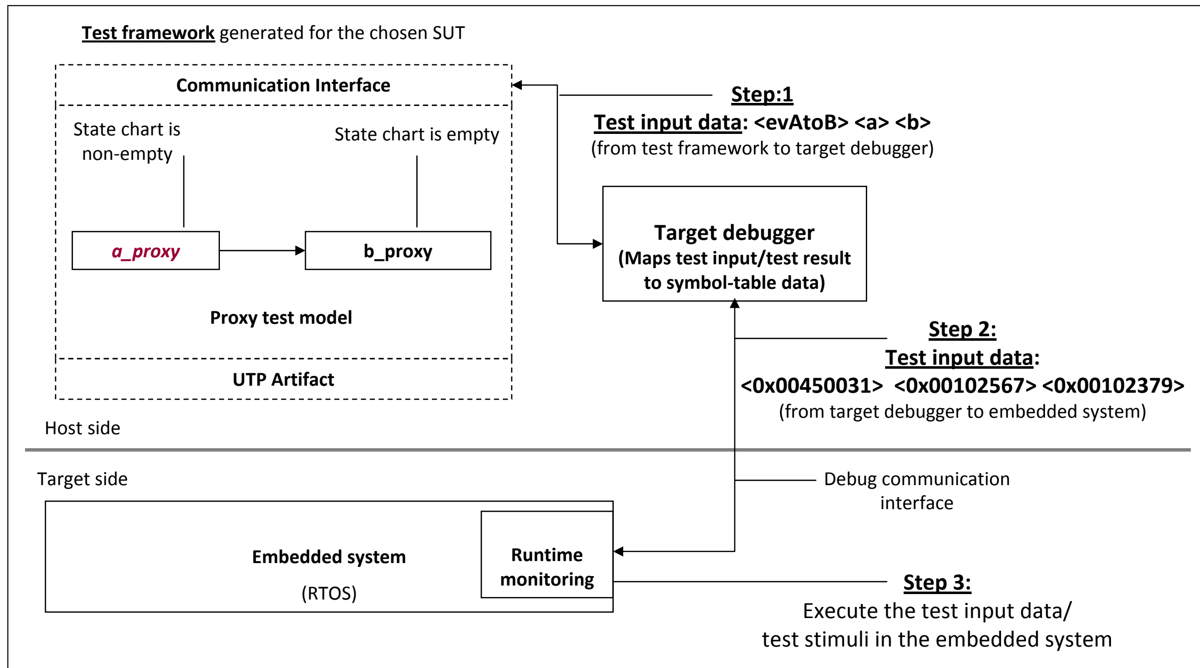



Figure 3.8: Steps involved in decoding the test input data with the target debugger

in the pre-defined format is <*evAtoB*> <*a*> <*b*>. Here, the *event*, *source*, *destination* in the test input data are *evAtoB*, *a* and *b* respectively. However, if the test input data is sent in the above format (step 1), from the test framework to the target monitor routine (Figure 3.7), then additional overhead is incurred (step 2 in Figure 3.7) by the monitoring routine to decode this test input data and map them to object addresses (e.g. available in symbol-table data). This is required to execute the test input data in the embedded software running on the target (step 3). For example, without any decoding function (already) at the host computer, the test input data is sent as <*evAtoB*> <*a*> <*b*> to the runtime monitor routine. Then additional monitoring overhead is incurred to decode this test input data and then map the test data to the respective object addresses, i.e., test data to be injected to the target decoded by the monitoring routine is <*0x00450031*> <*0x00102567*> <*0x00102379*> (step 2 and then step 3 in the target).

On the other hand, in the presence of a decoding function at the host computer, for example in the target debugger, additional overhead for decoding the trace data at the embedded system can be eliminated (Figure 3.8). The series of steps involved in decoding the test input data with the aid of decoding functionality, such as a target debugger, at the host computer is shown in Figure 3.8. Thus, the test data to be injected to the target is readily available as test input from the target debugger as shown in Figure 3.8 (step 2). This intermediate step to decode the test input data at the target debugger eliminates the additional runtime monitoring overhead at the target (only step 3 is in the target).

Similar reasoning is applicable while receiving the test results from the embedded system at the host computer. Thus, addressing challenge (3) pertaining to a minimally intrusive runtime monitoring methodology, decoding functionality such as that of a target debugger discussed above is proposed in this thesis. With the aid of a information marshalling and decoding agent at the host computer, namely the target debugger, additional monitoring overhead at the embedded system can be eliminated. Then the overhead involved in the embedded system will be (only) that of the software-based runtime monitoring routine to insert the test input data and convey the test results (i.e., the trace data) to the host computer via a debug communication interface.

**Mapping of test input data and test results at the target debugger**

It is clear that, in order to minimize the monitoring overhead in the embedded system the test input data (e.g. class names) can be mapped to symbol-table data (e.g. object address), already at the host computer. This is necessary to insert the test input data in the embedded system and execute the test input data. Similarly, the test results available as object addresses (i.e., symbol-table data), needs to be mapped to their respective high level names (e.g. class, event name). Then the test results can be interpreted at the host computer, by the test framework. In order to achieve this goal, a mapping of the high-level names (e.g. class names) to their object addresses needs to be created in an intermediary format, such as a XML file, at the host computer. This can be used by the target debugger to decode the test input data or the test results to be used by the monitoring routine or the test framework respectively. To realize this goal, the following methodology is proposed in this thesis.

For instance, consider that during the compilation of the (application) system code (MDD phase), a linker map file which comprises of symbol-table information, i.e., a file with a mapping of high-level/design-level names (e.g. classes, events) to object addresses is created at the host computer. By parsing the linker map file and the sources of the application, an intermediary file (e.g. a XML

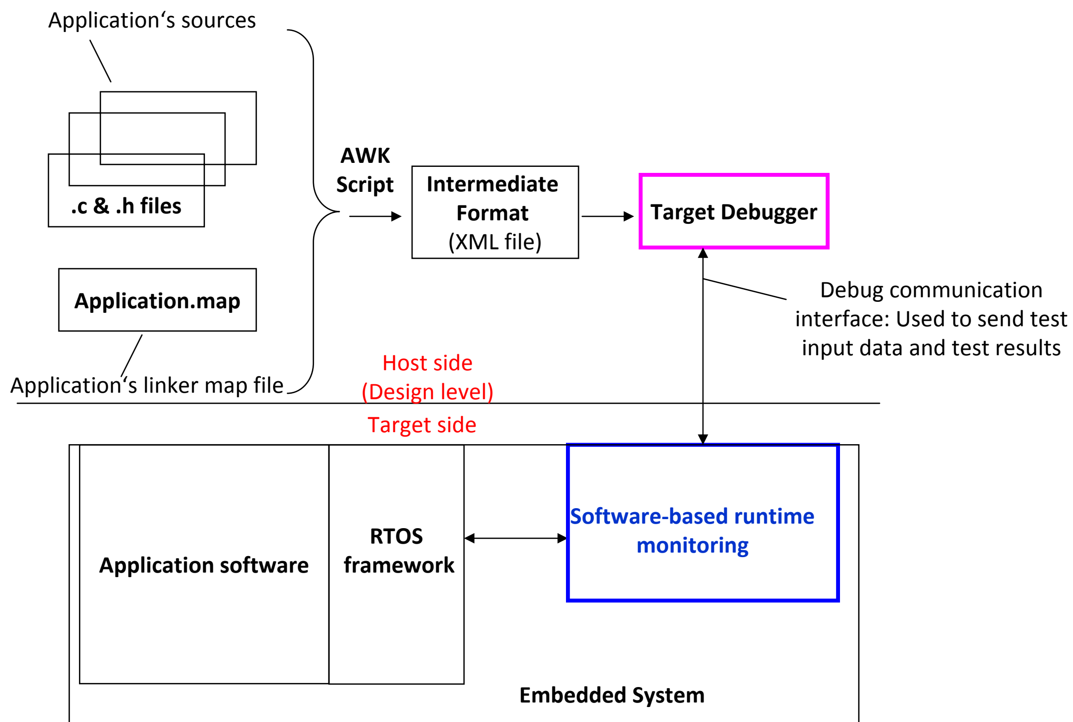



Figure 3.9: XML file creation at the host computer

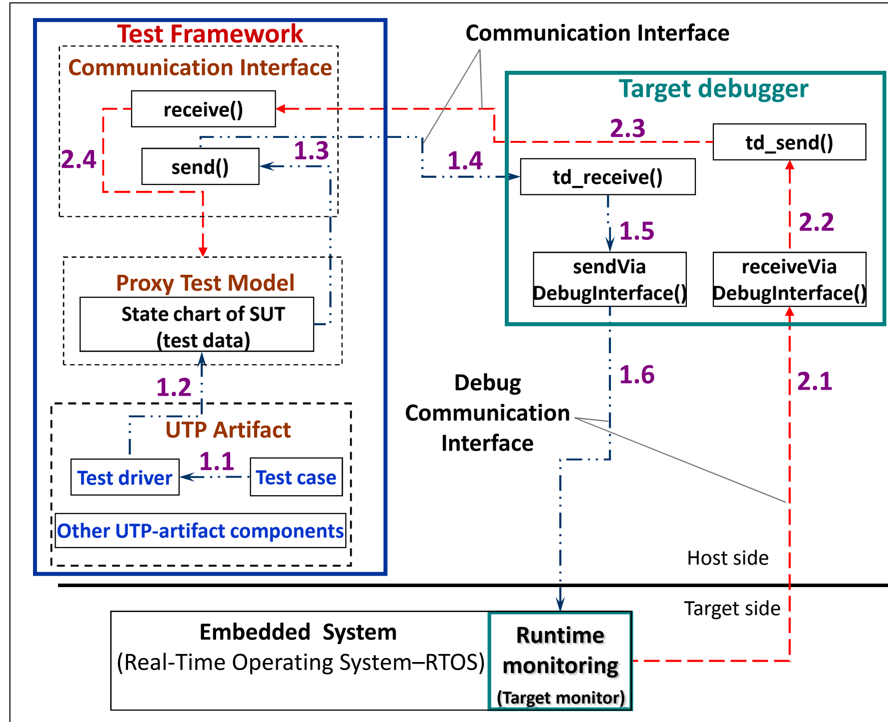
file) which stores this mapping of symbol-table data to high-level names, can be created at the host computer (Figure 3.9). Then, the target debugger can make use of the mapping available in the intermediary format (e.g. the XML file) to decode the test input data or the test results. This series of steps is illustrated in Figure 3.9.

Thus, by this methodology, significant overhead involved in sending the trace data back and forth between the host computer and the target system is avoided. This is possible by making use of the target debugger (Figure 3.9), the intermediary mapping to symbol-table data (i.e., the XML file) and the pre-defined frame format for test input and test results (Figure 3.5 and 3.6).

## 3.4 Executing model-based test cases in embedded systems

Addressing challenges (1), (2) and (3), an integrated model-based approach and test framework are introduced in the previous section. This section provides an outline on the steps involved in executing the model-based test cases, using the proposed approach.

Given a chosen SUT and the system design model, the test framework generation algorithm (proposed in Chapter 4) generates the necessary artifacts (i.e., the test framework) for executing model-based testing in embedded systems. Once the test framework is generated automatically, the test cases are specified manually in the test framework. In the proposed approach, the test case execution is carried out by the test framework at the host computer as illustrated in Figure 3.10. The test data is injected to the embedded system by the target debugger with the help of the target monitor. The test case execution results are sent by the runtime monitoring component to the host computer. The




Steps 1.1 to 1.6: executing the test cases
Steps 2.1 to 2.4: interpreting the test results

Figure 3.10: Series of steps for executing the model-based test cases in embedded systems

test results are then interpreted by the test framework at the host computer. Thus, the test case execution process comprises of two main steps, namely,

1. Executing the test cases on the host computer, illustrated as steps 1.1 to 1.6 in Figure 3.10.
2. Interpreting the test results, shown in steps 2.1 to 2.4 in Figure 3.10.

These two series of steps are indicated by different line formatting in Figure 3.10 and described below.

1. Executing the test cases
   - The test cases are specified (manually) by the user in the automatically generated test framework, in a MBT tool (e.g. [43]). This is shown as step 1.1 in Figure 3.10.
   - The test driver in the UTP artifact component of the test framework initiates the test case execution process, thereby injecting an event to trigger the SUT (step 1.2 in Figure 3.10).
   - On receiving the event injected by the test driver, the state chart of the SUT performs the respective action in the transition of the triggered event. The main functionality in the transition of the triggered event involves invoking the functions in the communication interface component with the test data corresponding to the triggered event (step 1.3). Note that the state chart of the SUT comprises of the test data corresponding to the injected event from the test case. The test (input) data is available in a pre-defined frame format as shown in Figure 3.5, in the transition of the triggered event (in the state chart of the SUT in the proxy test model).

- Thus the test data is conveyed to the target debugger by the communication interface component in the test framework (step 1.4).
- The target debugger receives this test input data, interprets it and sends the test input data to the embedded system via the APIs supported by the debug communication interface. These steps are illustrated as steps 1.5 and 1.6 in Figure 3.10. Thus the test data is conveyed to the embedded system by the target debugger via the target monitor in the embedded system.

2. Interpreting the test results

- The test stimuli/test input data sent by the target debugger is executed in the embedded system. The test input data sent comprises of an event or series of events to be executed corresponding to a test case. The results of the test case execution are sent by the target monitor to the target debugger via the debug communication interface. This is shown as step 2.1 in Figure 3.10. The trace data (e.g. comprising of the test result) sent by the target monitor is in the format illustrated in Figure 3.6.
- The incoming test results are received by employing the APIs supported by the debug communication interface. The target debugger then conveys the test data to the test framework using the corresponding communication interface component in the test framework (steps 2.2 and 2.3 in Figure 3.10).
- The test results sent by the target debugger to the test framework are in a pre-defined frame format as shown in Figure 3.6. The communication interface component (at the host) receives and decodes the incoming test results and invokes the respective functions in the proxy test model. This is illustrated as step 2.4. Based on this decoded data the respective functions are invoked in the proxy test model to mirror the test case execution process at the embedded system. During this step, in the background, the respective UML diagrams are drawn automatically in the test framework based on the invoked event reception operations and state changes in the proxy test model. Thus with the aid of the proxy test model in the test framework, the test results are visualized using UML diagrams in the MBT tool dynamically, in real time.

From the steps described above it is clear that the test cases are executed at the host computer. The test case execution process is stimulated by sending the test input data and receiving the test results with the help of the target monitor (in the embedded system) and the target debugger (at the host). The test case execution process at the embedded system is visualized in real time at the host computer using UML diagrams (e.g. sequence diagrams) with the aid of the test framework and the proposed approach.

Summarizing, in this chapter an introduction to the proposed integrated model-based approach and test framework for embedded systems is provided. The key idea of the integrated model-based approach is that, given the system design model and the chosen SUT (from the MDD phase) a test framework is generated for executing the model-based test cases in the embedded system (during the MBT phase).

In the next chapter, a formal notation for the system design model and the proposed test framework is introduced. By making use of the formal notation, a generic test framework generation algorithm is described with illustrative examples.

# Chapter 4

# Test Framework Generation

In this chapter, a generic approach towards generating a model-based test framework, at the host computer, for automating the test case execution process is described in detail. This pertains to challenge (2), which deals with a proposal for automatic generation of a model-based test framework for executing the model-based test cases in the embedded system. This chapter is organized as follows.

In section 4.1, a real-life embedded software engineering project comprising of a MIDI system analyzer is introduced (challenge (4)). This example is used to explain the generic notation for the system design model and the components of the test framework which are introduced in sections 4.2 and 4.3 respectively. The formalism is a pre-requisite to provide a generic algorithm for the test framework generation. Based on the formal notation, a generic, novel test framework generation algorithm is introduced in section 4.4 (challenges (1) and (2)). The various steps of the algorithm are described using examples from the MIDI system analyzer case study. A complexity analysis of the proposed algorithm is discussed in two perspectives, in section 4.5. This provides insights into the overhead parameters in the proposed approach (challenge(3)). The aforementioned theoretical estimates from the complexity analysis are recalled during an experimental analysis of the parameters in the complexity measures in chapter 5.

In this thesis, the prototype implemented for the proposed test framework is evaluated in embedded software application scenarios developed using UML. Therefore, in the example application scenarios, UML is used as the modeling language to specify the system design model and the test framework is generated using UML and UTP concepts. Examples based on UML notation (from the MIDI system analyzer case study) are cited when describing the generic notation introduced in this chapter.

## 4.1 MIDI system analyzer case study

The MIDI system analyzer example is introduced in this section before the formal notation, in order to provide a background on the requirements and goals of the MIDI system analyzer case study. This is because, the formal notation introduced in the next sections (sections 4.2 and 4.3) are explained based on examples from the MDD and MBT phases of the MIDI system analyzer case study.

The goal of this example application scenario is to develop and test a MIDI system analyzer embedded software application using the proposed integrated model-based approach and test framework. By this, the goals pertaining to challenge (4) which deal with the evaluation of the proposed approach

in a real-life embedded software engineering project is envisioned to be addressed.

MIDI [2], [28] is a music industry standard communication protocol that allows electronic musical instruments and computers to send instructions to each other. A MIDI keyboard is used for sending MIDI signals or commands to other devices connected to the same interface as the keyboard. MIDI devices communicate with each other using MIDI messages, which are transferred between the devices as a sequence of bytes. The main goal of this case study is to develop and test a MIDI system analyzer application scenario using the proposed approach. The embedded software application, thus developed, is envisaged to be executed on a resource constrained embedded system (such as the MCB1700 evaluation board [69]). The embedded system is connected to a MIDI keyboard, which generates MIDI signals on key press (Figure 4.1). Thus the main functionality envisioned, for the MIDI system analyzer application running on the embedded system, is to interpret the incoming MIDI signal and generate the corresponding sound (music note) in real time.

An embedded software application such as the MIDI system analyzer is chosen as a case study to evaluate the proposed approach, based on the following reasoning. Since the proposed approach deals with the actual execution of the model-based test cases in a resource constrained embedded system, an experimental setup for evaluating the prototype implementation is necessary. In other words, unlike developing simulation scenarios or evaluating the embedded software application in the host computer, the proposed approach envisages to execute the model-based test cases in real-life application scenarios. Therefore to provide a realistic approach towards real-life experimental evaluation, a MIDI system analyzer embedded software application is envisaged to be developed. Experimental setup of such a MIDI system analyzer case study provides the convenience of interacting with the embedded system, and enables the end-user to execute the model-based test cases on the actual embedded system.

This case study is developed using the integrated model-based approach (Figure 3.1) described in chapter 3. A mapping between the formal notation (introduced in this chapter) to examples from the MDD and the MBT phases of the MIDI system analyzer case study is elaborated in this chapter. Therefore, to place in context the examples from the design model developed during the MDD phase and the test framework automatically generated during the MBT phase for the MIDI system analyzer, an outline of the requirements for the MIDI system analyzer example are provided in this chapter. Given this background, the formal notation for the MDD and the MBT phases are introduced and explained with examples in section 4.2 and 4.3 respectively.

#### Requirements

Consider an example of a MIDI system analyzer to be implemented as an embedded software application using the MDD approach shown in Figure 3.1 (left side). The main functionality envisaged for the MIDI system analyzer application running on the embedded system is to interpret the incoming MIDI signals (e.g. from a key press on a musical keyboard) and generate the corresponding note, in real time. This requirement is illustrated in Figure 4.1. The incoming MIDI signals (e.g. from key press) are conveyed as bytes (e.g. data representing key press message), to the MIDI system analyzer application in the target.

The basic functional requirement envisaged to be realized in this MIDI system embedded software application example is the following. The MIDI system application software developed for the embedded system and running on the embedded system shall interpret the MIDI commands issued by the MIDI controller/keyboard. In other words, when a user presses a key on the keyboard, the incoming

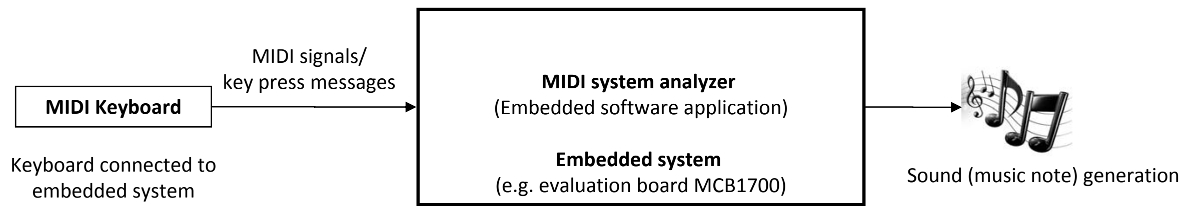



Figure 4.1: Requirements for the MIDI system analyzer case study

| **Command byte** (value=128) | **Key number** (value=60) | **Attack velocity** (value=100) |
|---|---|---|

Figure 4.2: MIDI protocol with a command byte for a key press action

MIDI sequence information must be decoded by the embedded software and a note must be played on the embedded system. Thus, on a key press in the MIDI keyboard the incoming command bytes (in the range of 128-255) should be decoded by the target. For a given sampling frequency all notes must be read and the sound needs to be generated.

The non-functional/performance aspects for this embedded software application are to meet the following requirements, namely: (a) the maximum latency/delay from keystroke to sound has to be less than 5 ms and (b) the maximum jitter (difference in delay) between the keystrokes and sound generation has to be less than 2 ms[1].

### MIDI command

MIDI bytes are split into two basic groups. They are command bytes in the range 0-127 (e.g. tuning the keyboard) and data bytes in the range 128-255 (e.g. key press on the keyboard) [2]. The aim of the MIDI system analyzer application is to decode only the key press messages (128-255) and filter any other messages. An example of a key press message is shown in Figure 4.2. A sequence of bytes "128 60 100", shown in Figure 4.2, indicates a message to turn on a note. The value "128" denotes that a note must be turned on, "60" indicates that a note corresponding to middle "C" should be played and "100" denotes how loud to play the note (i.e., in this case to turn ON the note). Hence corresponding to a key press on the keyboard, three messages are generated, namely a command byte (e.g. 128) followed by two data bytes (e.g. 60 and 100). For the aforementioned key press action (in Figure 4.2), the sequence of bytes for the key release action is "60 0". Here "60" denotes which key should be played and "0" denotes how loud to play the note (i.e., in this case to turn OFF the note). Thus, the sequence of bytes "128 60 100" followed by "60 0" indicates a key press action followed by the respective key release action.

The formal notation for the system design model and the test framework approach proposed in this thesis, are introduced in the forthcoming sections. Given the above background and the requirements for the MIDI system analyzer case study, examples from the MIDI system analyzer embedded software application, developed using the proposed approach (chapter 3, Figure 3.1), are used to explain the formal notation in detail.

[1] These values are the least ones to be matched, in order to be noticed by the human ear [118].

## 4.2 Model Driven Development phase

The formal notation for the system design model developed during the MDD phase is discussed in section 4.2.1 (challenge (2)). The notation is explained in detail with the help of examples from the MIDI system analyzer case study.

### 4.2.1 Software design model

Consider an embedded software system design model consisting of a set of associated modules, each with its respective set of states, attributes, operations, associations and dependencies. The design model of such a system is denoted as a package $P^{App} = M^{App}$ (Figure 4.3), where $M^{App}$ is defined as a set of modules $M^{App} = \{M_1^{App}, M_2^{App}, ..., M_n^{App}\}$ as shown in Figure 4.3. Each module ($M_n^{App}$) can be associated with other modules in the system design model. For example, $M_1^{App}$ is associated with $M_2^{App}$ and so on, as seen in Figure 4.3.

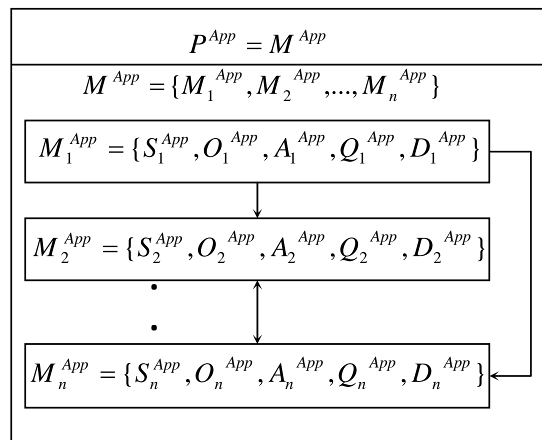



Figure 4.3: Simple embedded system software design model

Consider that the requirements and functionality of the MIDI system analyzer is realized using the MDD approach outlined in this thesis (MDD phase in Figure 3.1). The main functionality of the application is modeled using UML class diagrams and encapsulated in an UML package ($P^{App} = M^{App}$), such that $M^{App} = \{M_1^{App}, M_2^{App}, ..., M_n^{App}\}$.

The core functionality of the MIDI system analyzer can be grouped into the following, namely:

(a) Decoding the incoming message (key press message) and generating a valid MIDI command with the corresponding values for note generation.

(b) Interpreting the MIDI command and generating a note corresponding to the valid MIDI command (the value obtained from previous step).

(c) Generate the sound corresponding to the note information obtained from previous step.

The aforementioned functionality can be realized in three main modules. The corresponding three main classes envisaged for the design model of the MIDI system analyzer example are the *PreDecoder*, *MIDI_Interpreter* and *AudioSoundGen*.

The sequenced exchange of messages in the MIDI system example, envisioned during the design phase, is shown in Figure 4.4. In this figure, the two external hardware components connected to the

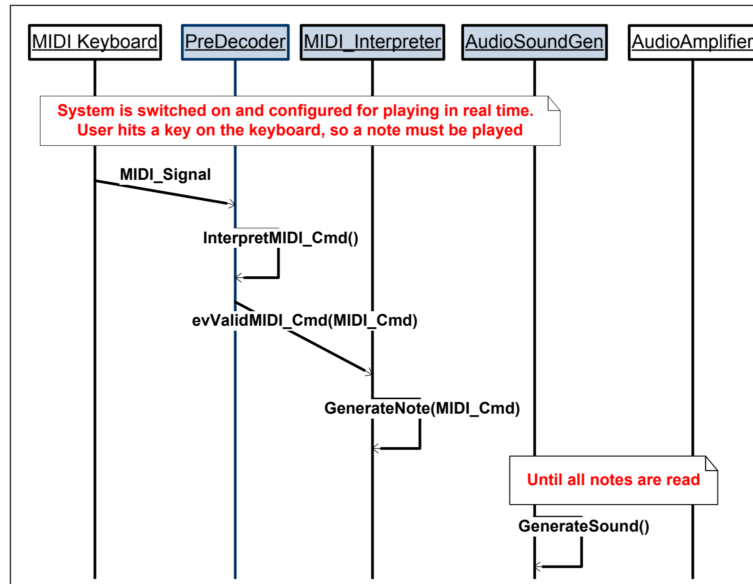



Figure 4.4: Message sequences in the MIDI system-design phase

embedded system are the MIDI keyboard and the audio amplifier. The three main classes envisaged for the MIDI system analyzer embedded software application, namely, *PreDecoder*, *MIDI_Interpreter* and *AudioSoundGen* are also shown in Figure 4.4. From Figure 4.4, it is clear that initially the system is switched on and configured for playing in real time. When a user hits a key on the keyboard a note must be played. As seen in Figure 4.4, the *PreDecoder* object is fed with the incoming data from the keyboard. It filters the redundant information from the data stream (e.g. message sequences other than key press). The *PreDecoder* object decodes the incoming MIDI message sequence and generates a valid MIDI command. The generated MIDI command is interpreted by the *MIDI_Interpreter* object which generates a corresponding note. The *AudioSoundGen* object samples the incoming notes at a given frequency and generates the corresponding sound, which is sent to the *AudioAmplifier*.

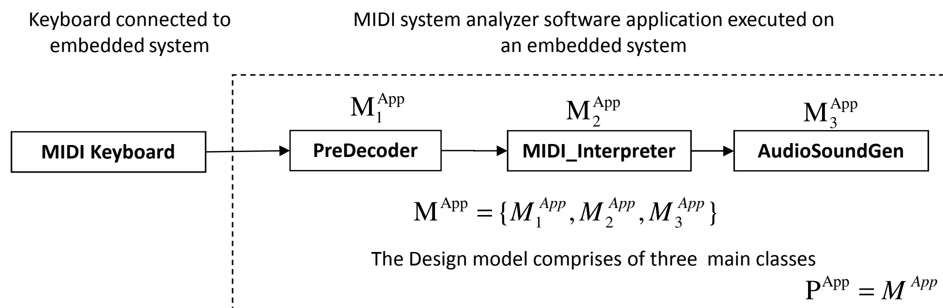



Figure 4.5: Overview of the design model for the MIDI system analyzer. The design model ($M^{App}$) is encapsulated in a package $P^{App}$

An overview of the design model for the MIDI system analyzer is shown in Figure 4.5. As seen in Figure 4.5, the MIDI keyboard is connected to an embedded system which runs the MIDI system

analyzer embedded software application. The design model for the MIDI system analyzer example (Figure 4.5) comprises of three main associated classes. The *PreDecoder* class ($M_1^{App}$) is responsible for decoding the incoming bytes and generating a valid MIDI message. The *MIDI_Interpreter* class ($M_2^{App}$) interprets the MIDI message and generates a note. The *AudioSoundGen* class ($M_3^{App}$) is responsible for generating the sound corresponding to the note. In this example the design model is represented as $M^{App} = \{M_1^{App}, M_2^{App}, M_3^{App}\}$.

Each module $M_n^{App}$ is represented as a tuple $M_n^{App} = \{S_n^{App}, O_n^{App}, A_n^{App}, Q_n^{App}, D_n^{App}\}$, as seen in Figure 4.3. The elements of the tuple are finite sets, where

- $S_n^{App}$ denotes the finite set of all states $s_k$, transitions $t_l$ and events $e_m$ respectively in the given class $M_n^{App}$. Then $S_n^{App} = \{s_k, t_l, e_m\}$. The state diagram with these states, transitions and events is represented as $S_n^{App(s,t,e)}$.
- $O_n^{App}$ denotes the set of all operations in every module. The operations in each module consist of an operation body represented as $O_{nf}^{App}$, i.e., $\forall O_{j=1...f}^{App}\ \exists O_{jf}^{App}$.
- $A_n^{App}$ represents the attributes in every module.
- $Q_n^{App}$ is the set of associations for each module.
- $D_n^{App}$ represents the dependencies for each module.

In the proposed approach, a test framework is generated for a chosen SUT. For example, the SUT can be any module $M_n^{App}$ (e.g. class). In this case, the SUT may also be referred as $M_{sut}^{App}$.

The modules for the design model ($M_1^{App}, M_2^{App}, ..., M_n^{App}$) are represented using UML class diagrams. For example, in the MIDI system analyzer case study the *PreDecoder* class ($M_1^{App}$), the *MIDI_Interpreter* ($M_2^{App}$) class and the *AudioSoundGen* class ($M_3^{App}$) are the three main classes in the system design model (Figure 4.4). Thus, for this example, $M^{App} = \{M_1^{App}, M_2^{App}, M_3^{App}\}$. Similarly, the various elements described above for the system design model can be represented by means of UML elements for a given project.

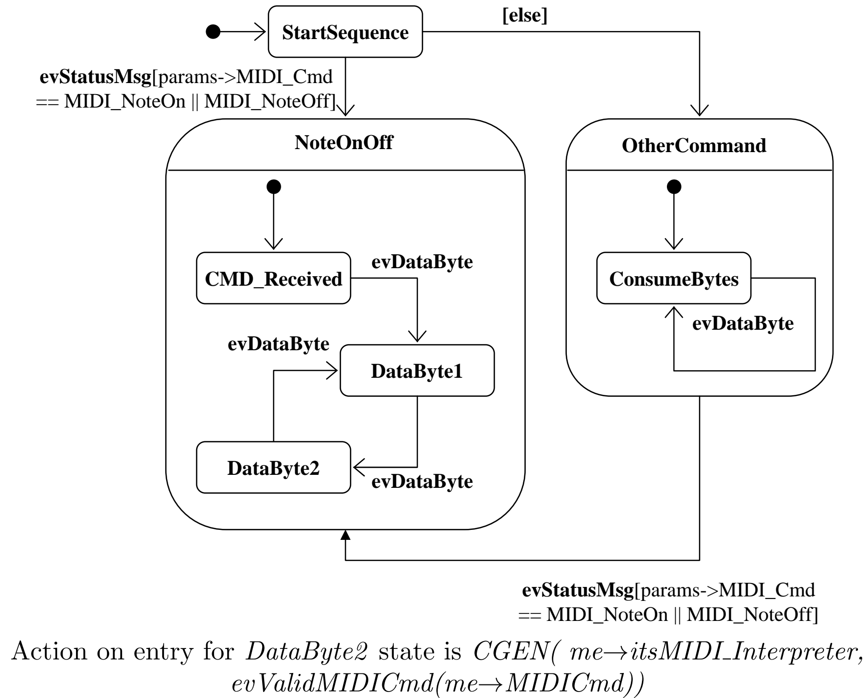
A typical mapping between the components of each module ($S_n^{App}, O_n^{App}, A_n^{App}, Q_n^{App}, D_n^{App}$) and their respective representation as an UML element is discussed below. A mapping between the states, transitions, events ($S_n^{App} = \{s_k, t_l, e_m\}$) and the resulting state diagram ($S_n^{App(s,t,e)}$) of the individual classes to their respective UML elements (i.e., UML state charts) is explained in detail in section 4.2.2. A mapping between the remaining components of each module, namely $O_n^{App}$, $A_n^{App}$, $Q_n^{App}$, $D_n^{App}$ to their respective UML elements is described in section 4.2.3.

### 4.2.2 Detailed behavior modeling using UML state charts

Based on the above discussion it is clear that the individual modules of the overall system design model can be specified using UML class diagrams. In turn, the detailed behavior modeling of a class can be represented by using UML state charts.

#### 4.2.2.a State chart of *PreDecoder* class

Consider the detailed modeling of the reactive behavior of the *PreDecoder* class as shown in Figure 4.6. The main functionality of this class is to filter the incoming bytes, other than the ones representing




Action on entry for *DataByte2* state is *CGEN( me→itsMIDI_Interpreter, evValidMIDICmd(me→MIDICmd))*

Figure 4.6: State chart of *PreDecoder* class ($S_1^{App(s,t,e)}$)

a key press/key release message. It also needs to aggregate the incoming bytes corresponding to the key press and key release messages and generate a valid MIDI message. This functionality is achieved by the two states *NoteOnOff* and *OtherCommand* respectively in the state chart of the *PreDecoder* class. Once a key press message is received (e.g. value=128), the state chart expects the next two data bytes (e.g. values=60 and 100), of the MIDI message (refer to MIDI message format in Figure 4.2). Immediately following this key press message, the sequence of bytes representing the corresponding key release message (e.g. values=60 and 0) is expected. Thus, the sequence of messages expected by the *PreDecoder* class is a command byte followed by two data bytes for a key press message. This needs to be immediately followed by the respective key release message with two data bytes, for a valid MIDI message to be generated. This is in accordance with the MIDI protocol format as seen in Figure 4.2. Thus, the *PreDecoder* class is responsible for decoding the incoming data bytes representing the MIDI message.

The distinction between a key press message and the other messages is made by the parameter *MIDI_Cmd* of the event *evStatusMsg(MIDI_Cmd).* Thus, only when the value of the parameter *MIDI_Cmd* is equal to *MIDI_NoteOn* (e.g. value=128) or *MIDI_NoteOff* (e.g. value=144), the state *NoteOnOff* is entered, i.e., *MIDI_Cmd=MIDI_NoteOn*||*MIDI_NoteOff* as seen in Figure 4.6. In other cases, the incoming data bytes are consumed and ignored by the application. This functionality is shown in the *OtherCommand* state in Figure 4.6. Once the incoming bytes are processed by the *NoteOnOff* state, i.e., once it receives an *evStatusMsg(MIDI_Cmd)* message followed by two *evDataByte(Byte)* messages, a valid *MIDI_Cmd* message can be generated. This functionality to generate a valid MIDI command is implemented in the entry action of the state *DataByte2*. The entry action of

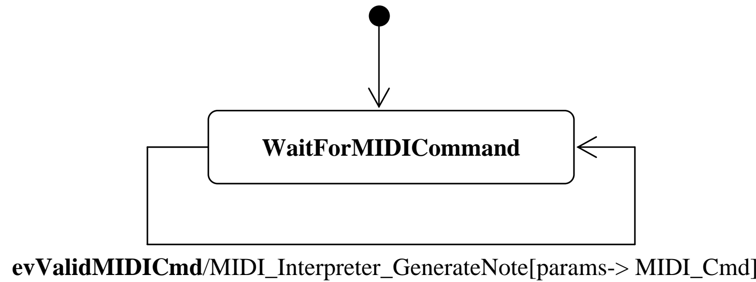



Figure 4.7: State chart of *MIDI_Interpreter* class ($S_2^{App(s,t,e)}$)

this state, namely *CGEN*[2]*( me→itsMIDI_Interpreter, evValidMIDICmd(me→MIDICmd))*, is shown in Figure 4.7. Thus, an *evValidMIDICmd(me→MIDIiCmd)* event is generated to the *MIDI_Interpreter* class, once the incoming key press message is decoded by the *PreDecoder* class. The value of the parameter *MIDICmd* corresponds to a valid note.

#### Mapping between the state chart of *PreDecoder* class and the formal notation

Consider the states, transitions and events in the state chart of the *PreDecoder* class ($M_1^{App}$) represented as a set $S_1^{App} = \{s, t, e\}$ (Figure 4.6). Here, $s$={*StartSequence, NoteOnOff, OtherCommand, CMD_Received, DataByte1, DataByte2, ConsumeBytes*} is defined as the set of states with $|s|$=7. Similarly, $e$={*evStatusMsg(MIDI_Cmd), evDataByte(Byte)*} is defined as the set of events with $|e|$=2. In this example the total number of transitions is $|t|$=10. Then, the state chart of the *PreDecoder* class ($M_1^{App}$) can be represented as $S_1^{App(s,t,e)}$ (Figure 4.6).

#### 4.2.2.b State chart of *MIDI_Interpreter* class

The state chart of the *MIDI_Interpreter* class, by default, waits for a valid MIDI command. This is indicated by the event reception *evValidMIDICmd()* for the only transition in the *MIDI_Interpreter* class, as shown in Figure 4.7. Once a valid MIDI command is received, the *MIDI_Interpreter* class generates a note, based on the parameters of the incoming valid MIDI command.

#### Mapping between the state chart of *MIDI_Interpreter* class and the formal notation

The state chart of the *MIDI_Interpreter* class (Figure 4.7), can be represented as $S_2^{App} = \{s, t, e\}$. The set of states is defined as $s$={*WaitForMIDICommand*} with $|s|$=1, the set of events is defined as $e$={*evValidMIDICmd(MIDI_Cmd)*} with $|e|$=1 and the total number of transitions is represented as $|t|$=2. Then, the state chart of the *MIDI_Interpreter* class ($M_2^{App}$) is denoted as $S_2^{App(s,t,e)}$ (Figure 4.7).

### 4.2.3 Mapping between formal notation and UML elements

A mapping between the components of a state diagram (in a given module represented as an UML class diagram) and a typical UML state chart has been explained in section 4.2.2. In this section, a

[2]The macro *CGEN(destination, event(parameters))* is used to generate an event, as per the rules of the modeling tool (Rhapsody [42]) used in the evaluated prototype and examples.

mapping between the remaining components of each module, namely $O_n^{App}$, $A_n^{App}$, $Q_n^{App}$, $D_n^{App}$ and their respective UML elements is described. The aforementioned elements can be mapped to UML elements such as operations, attributes, associations and dependencies respectively.

Consider the implementation of the *PreDecoder* class in the MIDI system analyzer, in a MDD tool, Rhapsody [42]. For the *PreDecoder* class, the set of operations ($O_1^{App}$), attributes ($A_1^{App}$), associations ($Q_1^{App}$) and dependencies ($D_1^{App}$) are as shown in Figure 4.8. For example, $O_1^{App}$ =

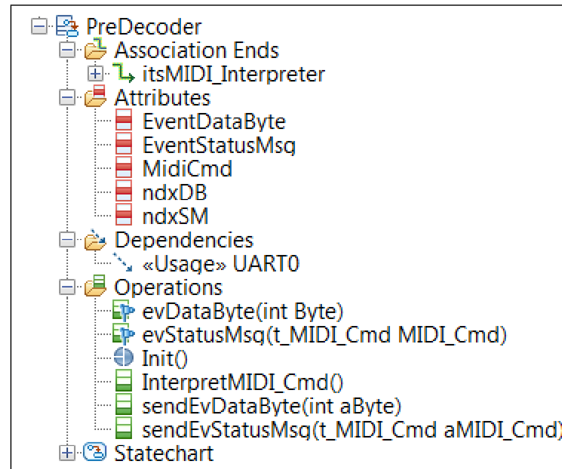



Figure 4.8: Screenshot showing the implementation of the *PreDecoder* class in a MDD tool [42]

$\{evDataByte(), evStatusMsg(), Init(), InterpretMIDI_Cmd(), sendEvDataByte(), sendEvStatus$-$Msg()\}$, i.e., $|O_1^{App}|$=6. Similarly, $|A_1^{App}|$=5 (attributes), $|Q_1^{App}|$=1 (association ends) and $|D_1^{App}|$=1 (dependencies) for the *PreDecoder* class. Note that in the MDD tool used, each event in a class must be explicitly consumed/realized in the state chart using event reception-operations. In other words, corresponding to an event, there exists a respective operation to realize the event in a given class. Therefore, for the *PreDecoder* class the event reception operations corresponding to the events *evStatusMsg(), evDataByte()* are highlighted with a flag symbol, as seen in Figure 4.8. Similarly, the initializer operation (e.g. used to assign initial values to variables), is shown distinctly with a disc symbol in the MDD tool used. For the *PreDecoder* class, this is seen in the *Init()* operation (Figure 4.8).

Thus in the design model, UML class diagrams are used to specify the functionality of the respective modules. These class diagrams (representing the modules) are grouped in an UML package. Detailed modeling of the functionality of each class is specified by means of UML state chart diagrams with states and transitions between them. The remaining elements in the system design model, namely, operations, attributes, associations and dependencies for a given module can also be represented as their respective UML elements as discussed above. Thus, the notations for a given system design model (seen in Figure 4.3), can be mapped to UML elements as discussed in this section. Mapping between the formal notation and the UML elements for the system design model is shown in Table 4.1. However, note that the generic notation can also be mapped to other modeling alternatives used in tools such as Matlab/Simulink [68], LabView [60], etc.

Table 4.1: Mapping between the formal notation and UML elements for the system design model

| **Formal Notation** | **UML Element** |
|---|---|
| $P^{App}$ | Package |
| $M_n^{App}$ | Class |
| $S_n^{App(s,t,e)}$ | State chart |
| $O_n^{App}$ | Operation |
| $A_n^{App}$ | Attribute |
| $Q_n^{App}$ | Association |
| $D_n^{App}$ | Dependency |

## 4.3 Model Based Testing phase

A generic notation for the system design model has been introduced in the previous section. This is provided as a base for addressing challenges (1) and (2). These challenges pertain to the application of an integrated model based approach and generic test framework, for executing the model-based test cases in the embedded system. The generic test framework is generated automatically in the proposed approach, based on the given system design model and the chosen SUT. Therefore, to provide a generic notation for the test framework during the MBT phase, a generic notation for the system design model has been provided in section 4.2.

Similarly, a generic notation for the test framework, which is envisaged to be automatically generated using a test framework generation algorithm, proposed in this thesis, is introduced in section 4.3.1. The generic notation pertaining to the components of the test framework are explained with the help of examples from the MIDI system analyzer case study. The test framework is generated based on the chosen SUT and the system design model. Therefore, while explaining the generic notation for the individual elements in the test framework, examples from the MIDI system analyzer are cited by choosing the *PreDecoder* class as the SUT.

### 4.3.1 Test framework

Given a design model $P^{App} = M^{App}$, the test framework proposed in this thesis for executing the model-based test cases in embedded systems, can be represented as a tuple $P^{Test} = \{M^{Test}, T^{Com}, T_{Arch}\}$. The formal notation corresponding to the test framework is shown in Figure 4.9. Similar to $P^{App}$, the test framework is encapsulated in an UML package element $P^{Test}$. The three components of the test framework (Figure 4.9) are *proxy test model* ($M^{Test}$), *communication interface* ($T^{Com}$) and *UTP artifact* ($T_{Arch}$). The definition, formal notation of the three components of the test framework and a description of their functionality are provided in this section.

For example, the test framework generated for the chosen SUT, i.e., the *PreDecoder* class in the MIDI system analyzer example is illustrated in Figure 4.10. It comprises of the three components for the test framework, namely, *proxy test model* ($M^{Test}$), *communication interface* ($T^{Com}$) and the *UTP artifact* ($T_{Arch}$) as seen in Figure 4.10.

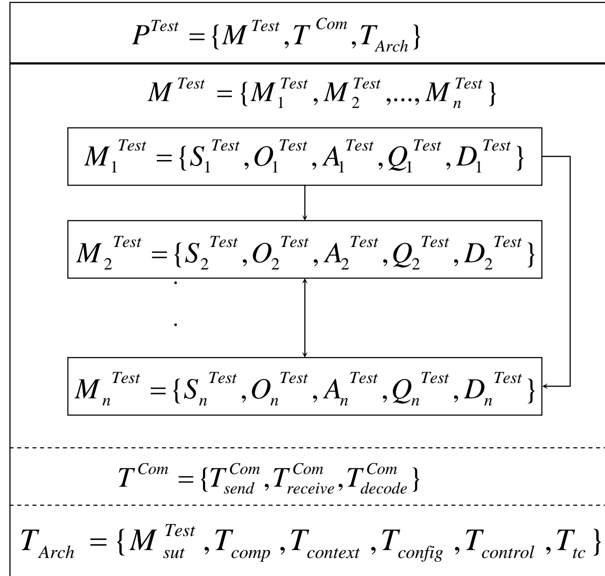



Figure 4.9: Test framework notation

### 4.3.2 Proxy Test Model

The *proxy test model* component of the test framework consists of a corresponding test module with respect to each module in the association end of the chosen SUT in the system design model. However, the proxy test model consists of only an abstract design model with no functionality inherited/derived from the system design model. In other words, corresponding to each module $M_n^{App}$ in the association end of the chosen SUT in the design model, a module $M_n^{Test}$ is created in the proxy test model. Thus, the proxy test model is represented as $M^{Test} = \{M_1^{Test}, M_2^{Test}, ..., M_n^{Test}\}$ and is specified as a set of UML class diagrams.

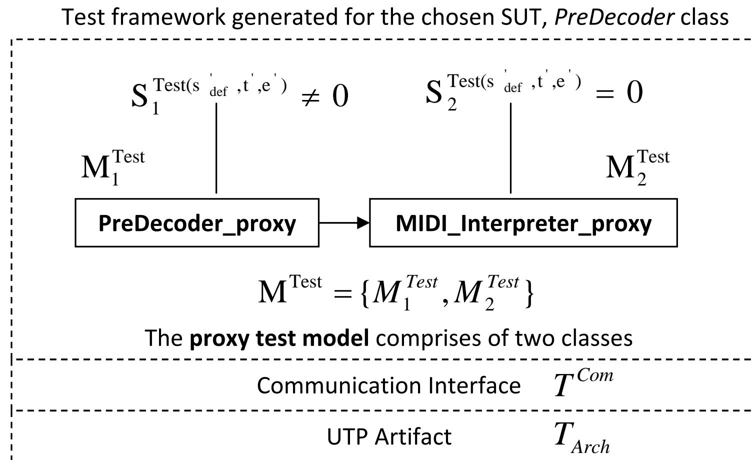



Figure 4.10: Test framework generated for the chosen SUT, *PreDecoder_proxy* class

Thus, for the chosen SUT, i.e., the *PreDecoder* class (Figure 4.5), the proxy test model comprises of two classes namely *PreDecoder_proxy* and *MIDI_Interpreter_proxy* as seen in Figure 4.10. While

the system design model comprises of three main classes, the proxy test model comprises of only two classes. This is because the chosen SUT, i.e., the *PreDecoder* class is associated with only one class from the system design model, i.e., the *MIDI_Interpreter* class (Figure 4.5).

For the test cases (e.g. manually specified or automatically generated) in the test framework, the test input data for the chosen SUT is available in the state chart of the SUT in the proxy test model. However the state chart of the proxy classes is empty, for all other associated classes of the SUT in the proxy test model. For example, the state chart of the *PreDecoder_proxy* class comprises of the test input data and the functionality to convey the test data using the communication interface component of the test framework. Whereas, the state chart of the other proxy classes, in this example, *MIDI_Interpreter_proxy* is empty in the proxy test model. This is also illustrated in Figure 4.10.

**Mapping of the elements in the proxy test model to a formal notation**

Consider a mapping of the elements in the proxy test model to a formal notation. Similar to the notation used in the specification of the design model, the proxy test model automatically generated for the chosen SUT can be denoted as $M^{Test} = \{M_1^{Test}, M_2^{Test}, ..., M_n^{Test}\}$ (Figure 4.9). In the chosen example, $M^{Test} = \{M_1^{Test}, M_2^{Test}\}$ (Figure 4.10), where, $M_1^{Test}$ is *PreDecoder_proxy* class and $M_2^{Test}$ is *MIDI_Interpreter_proxy* class. In line with the mapping for the design model notation to UML elements, the various components in the test framework notation are represented using UML elements as seen in Table 4.2. Each module $M_n^{Test}$ is represented as a tuple $M_n^{Test} =$

Table 4.2: Mapping between the formal notation and UML elements for the test framework

| **Formal Notation** | **UML Element** |
|---|---|
| $P^{Test}$ | Package |
| $M_n^{Test}$ | Class |
| $S_n^{Test(s,t,e)}$ | State chart |
| $O_n^{Test}$ | Operation |
| $A_n^{Test}$ | Attribute |
| $Q_n^{Test}$ | Association |
| $D_n^{Test}$ | Dependency |
| $T^{Com}$ | Singleton object |
| $M_{sut}^{Test}, T_{driver}$ | Instance of type class |

$\{S_n^{Test}, O_n^{Test}, A_n^{Test}, Q_n^{Test}, D_n^{Test}\}$, as seen in Figure 4.9. The elements of the tuple are finite sets, where

- $S_n^{Test}$ denotes the finite set of states $s'_{def}$, transitions $t'_l$ and events $e'_m$ respectively in the proxy class $M_n^{Test}$. Then $S_n^{Test} = \{s'_{def}, t'_l, e'_m\}$. The state chart of the proxy class with these states, transitions and events is represented as $S_n^{Test(s'_{def},t',e')}$. However, $S_n^{Test(s'_{def},t',e')} \neq \emptyset$ only for the state chart of the SUT in the proxy test model. For all other classes, the state chart of the SUT is empty, i.e., $S_n^{Test(s'_{def},t',e')} = \emptyset$ (Figure 4.10).

For example, $S_1^{Test(s'_{def},t',e')} \neq \emptyset$ corresponding to $M_1^{Test}$ (*PreDecoder_proxy*) class, whereas for $M_2^{Test}$ (*MIDI_Interpreter_proxy*), $S_2^{Test(s_{def},t',e')} = \emptyset$ (Figure 4.10).

- $O_n^{Test}$ denotes the set of all operations possible in every module in the proxy test model. Here, $\forall O_j^{App} \in M_j^{App} \exists O_j^{Test}$ , whereas $O_{jf}^{Test} = \emptyset$ are true. This implies that for every operation in the design model, there exists a corresponding operation/function call in the proxy test model; however the operation function body is empty in the proxy test model. This means that though the proxy test model is created based on the design model, it inherits only the abstract structure of the design model and not its functionality. This is true for all the modules (e.g. classes) and their elements (e.g. operations) in the proxy test model of the automatically generated test framework. $O_n^{Test}$is represented by means of functions in the corresponding UML class diagrams representing the proxy test model $M^{Test}$.

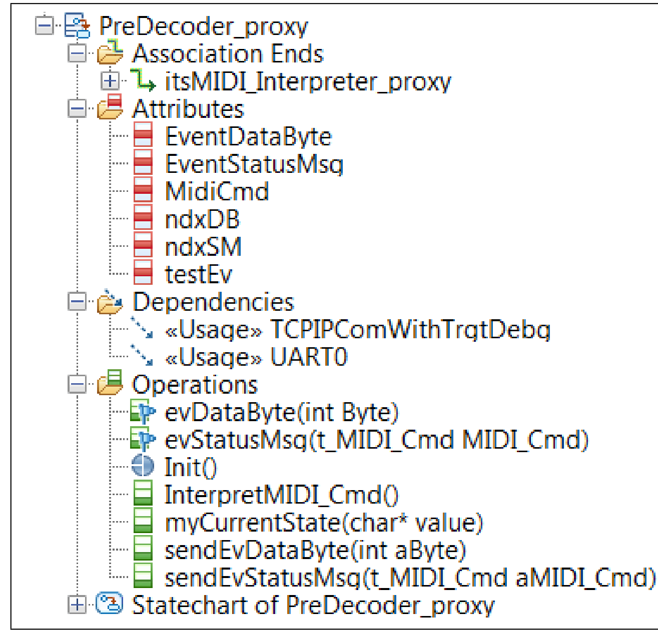



Figure 4.11: Screenshot of the (automatically generated) *PreDecoder_proxy* class in a MDD tool [42]

As seen in Figure 4.11, the operations from the design model for the *PreDecoder* class are copied to the *PreDecoder_proxy* class in the proxy test model. The set of operations for *PreDecoder* and the *PreDecoder_proxy* class are represented as $O_1^{App}$ and $O_1^{Test}$ respectively. In this example, $|O_1^{App}| = |O_1^{Test}|$ =7. However, the operation function body for these operations in the *PreDecoder_proxy* class is empty.

- $A_n^{Test}$, represents the attributes in every module. $A_n^{Test}$ is described as attributes in the UML class diagrams.

- $Q_n^{Test}$, is the set of associations for each module.

- $D_n^{Test}$, represents the dependencies for each module.

$A_n^{Test}, Q_n^{Test}$ and $D_n^{Test}$ are represented as attributes, associations and dependencies respectively in the class diagrams of the proxy test model. For the *PreDecoder_proxy* class, the attributes, associations

and dependencies are as shown in Figure 4.11. A detailed explanation on the generation of the components in the test framework (e.g. elements in the proxy test model) is provided during the description of the test framework generation algorithm in section 4.4.

### 4.3.3 Communication Interface

The communication interface component in the test framework is represented as a tuple $T^{Com} = \{T^{Com}_{send}, T^{Com}_{receive}, T^{Com}_{decode}\}$. $T^{Com}$ is responsible for two functionalities.

- First, $T^{Com}$ is used by the test framework to send and receive the test data between the target debugger and the host computer.
- Second, it encompasses the functionality to decode, interpret and invoke the respective functions based on the received message (e.g. test results) from the target debugger. This component in the test framework needs to be in its own thread of execution, in order to facilitate uninterrupted communication between the test framework and the target debugger.

Hence, in $T^{Com}$, the functionality to send, receive and decode the test data is represented by $T^{Com}_{send}$, $T^{Com}_{receive}$ and $T^{Com}_{decode}$ respectively. The communication interface component in the test framework can be implemented using alternatives for interprocess communication such as a TCP/IP communication interface (e.g. using Winsock [72]) and a Component Object Model (COM) [89] interface as seen in Figure 4.12. In the prototype evaluated in this thesis, the communication interface component is

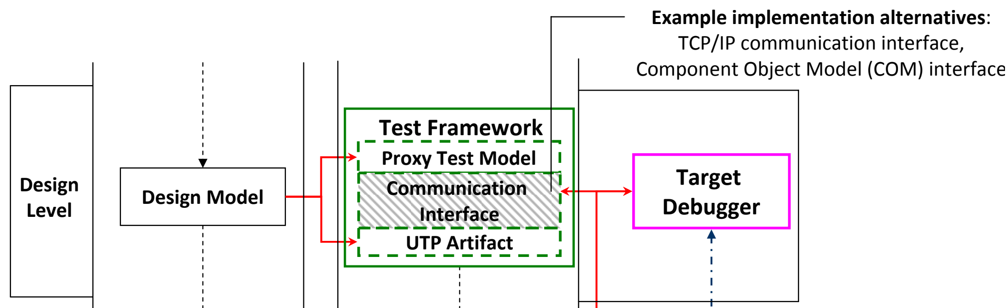



Figure 4.12: Communication interface component in the test framework

implemented as a TCP/IP communication interface. This component is added as a dependency to all the proxy classes generated in the proxy test model component of the test framework. For instance, the proxy test model makes use of the communication interface component to send the test data to the target debugger.

For the chosen SUT (*PreDecoder* class), the TCP/IP based communication interface component generated in the test framework is included as a dependency (to the module *TCPIPComWithTrgt-Debg()*) as shown Figure in 4.11.

### 4.3.4 UTP Artifact

The UTP artifact component of the test framework is envisioned to address the goals of this thesis pertaining to the usage of corresponding modeling languages for the MDD and the MBT phases in a generic test framework for embedded systems (challenges (1) and (2)). In this context, UML is

used for modeling and specification of the design model (MDD phase). UML and UTP artifacts are envisaged to be used for the test framework (MBT phase). The proxy test model component in the test framework is implemented (automatically generated) based on UML elements such as class diagrams and state charts.

The UTP artifact component in the test framework provides structural/infrastructure support towards executing the model-based test cases in embedded systems. As the name implies, this component is created based on the artifacts provided in the UTP. As described in chapters 2 and 3, the UTP profile introduces four concept groups namely, *test architecture*, *test behavior*, *test data* and *test time* [111], [123].

The entire set of UTP stereotypes for these concept groups can be recalled from chapters 2 and 3 (Table 2.1 and Table 3.1). Among the four concept groups, the *test architecture* group provides a set of concepts to specify the structural aspects of a test context covering the test components, the SUT and their configuration. The approach described in this thesis deals with automatically generating the test framework infrastructure and the architectural support at the host computer for executing model-based test cases in embedded systems. Hence, a concise subset of the UTP components (among the extensive set shown in Table 3.1) based on the test architecture concept group is chosen in the proposed approach. The UTP artifact ($T_{Arch}$) in the test framework ($P^{Test}$) is defined as a tuple of six elements, namely $T_{Arch} = \{M_{sut}^{Test}, T_{comp}, T_{context}, T_{config}, T_{control}, T_{tc}\}$. The UTP terminology from the *test architecture* concept group is mapped to the notation introduced for the UTP artifact component, as seen in Figure 4.13. The elements in the UTP artifact component are described below.

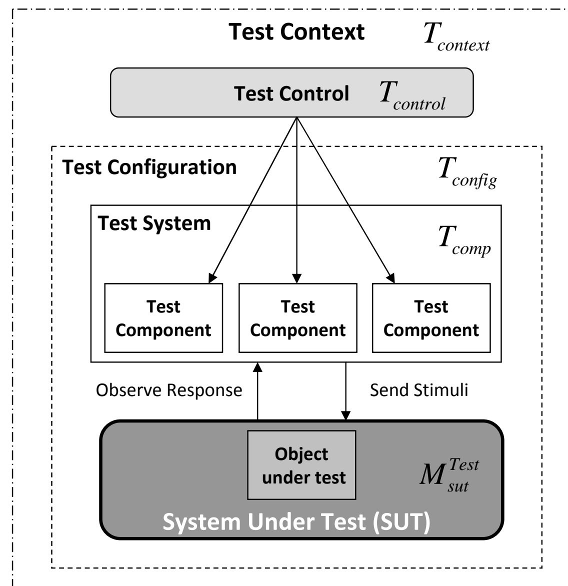



Figure 4.13: Mapping between UTP terminology and formal notation for UTP artifact

– The SUT ($M_{sut}^{Test}$) refers to a system, subsystem or component which is being tested. The SUT is stimulated via its public interface operations. The SUT may communicate with other component objects in a test system. For the MIDI system analyzer example, the chosen SUT is *PreDecoder_proxy* class.

- $T_{comp}$ denotes the finite set of all the interfaces/classes which may communicate among themselves or with the SUT. An example of a test component instance with which the chosen SUT *PreDecoder_proxy* class may communicate is the *MIDI_Interpreter_proxy* class. This is because the *PreDecoder_proxy* class is associated with the *MIDI_Interpreter_proxy* class in the proxy test model. This relation (i.e., association end) in turn has been derived for the proxy classes, based on the system design model encapsulating the *PreDecoder* and the *MIDI_Interpreter* class.

- The test configuration currently used for a UTP artifact is $T_{config}$. The set of all the test cases for the SUT under consideration is represented by $T_{tc}$. The collection of test cases together with the test configuration is termed as the test context $T_{context}$.

- A test control $T_{control}$ is a specification for the invocation of test cases within a test context. It is a technical specification of how the SUT should be tested with the given test context. In the proposed approach, the test control component, $T_{control}$, in the UTP artifact comprises of two control interfaces, namely, arbiter and scheduler.

  *Arbiter* is a predefined interface provided with the UTP. The purpose of the arbiter implementation is to determine the final verdict for a test case. This determination is done according to a particular arbitration strategy based on a default arbitration algorithm [111]. A test case or a test context use the arbiter to evaluate the test results and to assign its overall verdict.

  *Scheduler* is an entity within a test context that controls the running of the test cases. It keeps track of the creation and destruction of test components and gives instructions to the existing test components when to start executing a given test case. It communicates with the arbiter when the time is right to produce the verdict for a test case.

A common methodology for making use of an UML-based profile is to implement the profile as a set of stereotypes (e.g. in a MDD/MBT tool). The profile is then included in a given project. Then, the required or specific elements of the profile (e.g. UTP) can be used for a given UML element. For instance, this can be implemented by specifying the UML element with a particular stereotype from the profile. For example, a test component ($T_{comp}$) for the *PreDecoder_proxy* class is the *MIDI_Interpreter_proxy* class. Therefore, the *MIDI_Interpreter_proxy* class is specified (i.e., automatically generated by the test framework generation algorithm) with a stereotype $\ll TestComponent \gg$. This stereotype is available, for example in the UTP profile implemented in the MDD tool (e.g. Rhapsody [42], [43]).

A typical mapping of the formal notation for both the system design model and the components of the test framework to an example has been described in section 4.2 and 4.3 respectively. This is provided as a background and to place in context the steps involved in the generic algorithm for the test framework generation introduced in the next section (section 4.4).

## 4.4 Algorithm

An informal introduction to the test framework and its components is provided in chapter 3. In the previous section a generic and a formal notation for the test framework components is described with the help of a real-life embedded software application example. Given this background, in section 4.4.1, the test framework generation algorithm is presented and discussed in detail. This algorithm in turn invokes procedures to create the following components in the test framework, namely:

- State chart of the SUT (section 4.4.3)
- UTP artifact component and (section 4.4.4)
- Communication interface component (section 4.4.5)

All the algorithms are explained in depth with the help of examples from the MIDI system analyzer case study. Thus, challenge (2) pertaining to a generic test framework for executing the model-based test cases in the embedded system is addressed by the introduction of the generic notation and the test framework generation algorithm. Note: A prototype implementation of the proposed approach and the test framework generation algorithm is developed for experimental evaluation. The test framework generation algorithm is implemented in the programming language Java with the aid of the APIs [88] in the MDD/MBT tool, Rhapsody. The examples cited below from the MIDI system analyzer case study are generated by a prototype implementation of the proposed algorithm.

### 4.4.1 Test framework generation

In this section, a generic test framework generation algorithm envisioned to address the goals of challenge (2) is presented. This pertains to a proposal for the generation of a test framework at the host computer for executing the model-based test cases in the embedded system.

The test framework generation algorithm (Algorithm 1) takes as input the given system design model $P^{App} = M^{App}$ and the chosen SUT $M_{sut}^{App}$ (selected by the user) for which the test framework has to be generated. The algorithm generates a test framework $P^{Test} = \{M^{Test}, T^{Com}, T_{Arch}\}$. In the approach proposed in this thesis, a test framework is generated corresponding to a chosen SUT (from the design model). For example, the SUT can be any module $M_n^{App}$ (e.g. class). In this case, the SUT may also be referred as $M_{sut}^{App}$.

Consider the system design model for the MIDI system analyzer (Figure 4.5) and the chosen SUT ($M_{sut}^{App}$), i.e., the *PreDecoder* class (Figure 4.8) as input for this algorithm. The test framework generated corresponding to this input comprises of three components namely, the proxy test model, the communication interface and the UTP artifact (Figure 4.10). Examples based on the test framework automatically generated for the *PreDecoder* class (using a prototype implementation of the algorithm 1) are cited while explaining the various steps of the algorithm 1 below.

1. In the first step, the algorithm determines if the given application's package containing the system design model is not empty (line:1). If the UML package element $P^{App}$ denoting the system design model is not empty, then the algorithm creates a test framework package $P^{Test}$ (line:2). This package is used to accommodate the elements in the test framework namely, $M^{Test}$, $T^{Com}$ and $T_{Arch}$ which are intended to be created in the next steps (line:3-28) of the algorithm. If $P^{App}$ is empty, then the algorithm skips further processing (lines:3-28) and returns (line:29-31).

   For the chosen SUT (*PreDecoder* class), the corresponding system design model encapsulated in a package $P^{App}$ is non-empty (Figure 4.5). Hence, a test framework package *TPkg_TCon_PreDecoder_proxy* is created by the algorithm to accommodate the elements in the test framework as shown in Figure 4.14.

**Algorithm 1** : Test framework generation

**Input:** System design model $P^{App} = M^{App}$ and $M^{Test}_{sut}$
**Output:** Test framework $P^{Test} = \{M^{Test}, T^{Com}, T_{Arch}\}$

1: **if** $P^{App} \neq 0$ **then**
2: create $P^{Test}$ ;test framework package
3: **for** $\forall Q^{App}_n \in M^{Test}_{sut}$ **do**
4: **for** $\forall M^{App}_n \in Q^{App}_n$ **do**
5: $M^{Test}_n \leftarrow M^{App}_n$ ;create in $P^{Test}$
;for all association ends in the chosen SUT
;for all classes in this association end
;create a proxy class in the proxy test package
6: $O^{Test}_n \leftarrow O^{App}_n$ ;Operations $\forall O_i \in M_i \exists O^{Test}_i$ and $O^{Test}_{if} \neq 0$
7: $A^{Test}_n \leftarrow A^{App}_n$ ;Attributes
8: $Q^{Test}_n \leftarrow Q^{App}_n$ ;Associations
9: $D^{Test}_n \leftarrow D^{App}_n$ ;Dependencies
;copy operations, attributes, associations and dependencies from the class to the proxy
;class created in the previous step. For operations, only the function prototype (and not
;the definition) is copied.
10: **if** $M^{App}_n = M^{App}_{sut}$ **then**
11: $S^{Test(s'_{def},t',e')}_n \leftarrow createSUTStateChart(M^{App}_{sut}, S^{App(s,t,e)}_n)$
;if the processed class module is the SUT, a procedure *createSUTStateChart()* is invoked.
12: **else**
13: $S^{Test(s,t,e)}_n$ is created as an empty state chart
;if the processed class module is not that of the chosen SUT, then an empty state chart
;is created in the proxy class corresponding to this module
14: **end if**
15: **for** $\forall M^{App}_n \in P^{App}$ **do**
16: invoke procedure *generateEventConsumedOnTarget(*$M^{App}_n$*)* to create *generateEventConsumed()* operation in all proxy classes.
;this function is used to create the *generateEventConsumed()* operation for all the proxy
;classes.
17: **end for**
18: **end for**
19: $T_{Arch} \leftarrow createUTPArtifact(P^{App}, P^{Test}, M^{App}_{sut})$
20: **if** ($T_{Arch}$ =Error) or ($|T_{Arch}| = \emptyset$) **then**
21: GOTO line:30 ;procedure to create UTP artifact returned an error
22: **end if**
23: $T^{Com} \leftarrow createComObject(P^{App}, P^{Test}, M^{App}_{sut})$
24: **if** ($T_{Com}$ =Error) or ($|T_{Com}| = \emptyset$) **then**
25: GOTO line:30 ;procedure to create the communication interface returned an error
26: **end if**
27: **end for**
28: **return** $P^{Test} = \{M^{Test}, T^{Com}, T_{Arch}\}$ ;return the automatically generated test framework.
29: **else**
30: **return** Error ;report an error
31: **end if**

2. In the next steps (line:3-18), the proxy test model component of the test framework is created. For each association ($Q^{App}_n$) in the chosen SUT $M^{Test}_{sut}$ (line:3), the module(s) $M^{App}_n$ in the association are determined (line:4).

   In the chosen SUT ($M^{App}_1$ =*PreDecoder* class) there is one association end, as seen in Figure 4.8.

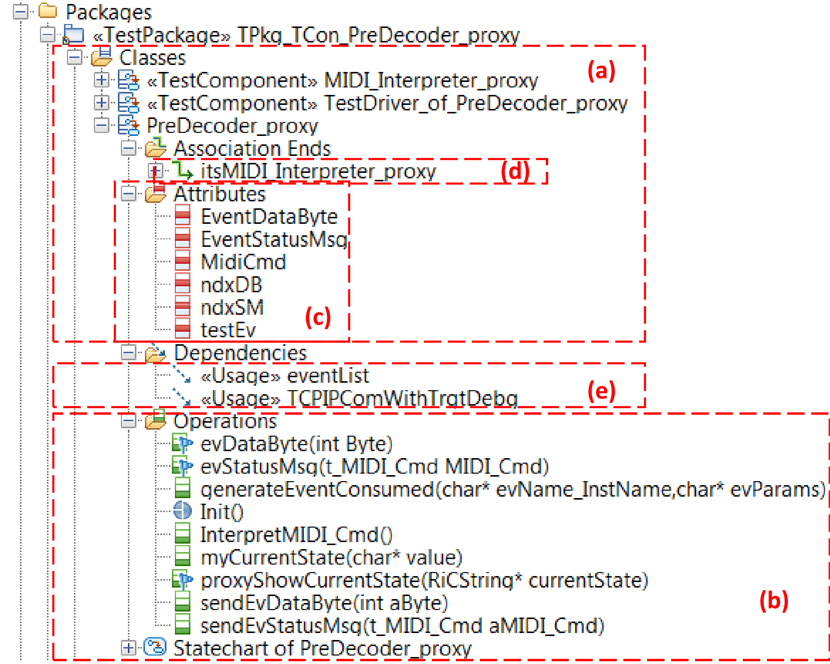



Figure 4.14: Proxy test model generated for the chosen SUT (*PreDecoder* class). (a) Proxy classes, (b) operations, (c) attributes, (d) association ends and (e) dependencies.

Therefore $|Q_1^{App}|$=1, i.e., $Q_1^{App}$ ={*itsMIDI_Interpreter*}. The corresponding class participating in this association end (*itsMIDI_Interpreter*) is the *MIDI_Interpreter* class ($M_2^{App}$).

For each module $M_n^{App}$ in the association $Q_n^{App}$ (line:4), the following steps are performed.

- For every module $M_n^{App}$ in the association end of the chosen SUT (in the design model package $P^{App}$), a corresponding proxy module $M_n^{Test}$ is created (line:5) in the proxy test package $P^{Test}$.

  For the participating class *MIDI_Interpreter* ($M_2^{App}$) in the association end (*itsMIDI_Interpreter*) of the chosen SUT i.e., the *PreDecoder* class (Figure 4.8), a corresponding proxy class *MIDI_Interpreter_proxy* ($M_2^{Test}$) is created in the proxy test package *TPkg_TCon_PreDecoder_proxy* (Figure 4.10, Figure 4.14).

  The participating classes in the association end are associated with the chosen SUT. Hence, a proxy class corresponding to the chosen SUT is also created in the test framework.

  For example, the participating class *MIDI_Interpreter* ($M_2^{App}$) in the association end (*itsMIDI_Interpreter*) is associated with the chosen SUT i.e., the *PreDecoder* class (Figure 4.8). Hence, a corresponding proxy module is created for the chosen SUT in the test framework. Thus, corresponding to the chosen SUT i.e., the *PreDecoder* class, a proxy module *PreDecoder_proxy* is created in the test framework (Figure 4.11, Figure 4.14).

- Next, the operations ($O_n^{App}$) of the module $M_n^{App}$ participating in the association end of the SUT are copied (line:6) to the proxy module $M_n^{Test}$ (newly created in the previous step) in the proxy test package $P^{Test}$.

  Consider the set of operations ($O_1^{App}$) for the *PreDecoder* class ($M_1^{App}$) in the system design

model, where $O_1^{App} = \{evDataByte(), evStatusMsg(), Init(), InterpretMIDI_Cmd(), send$-$EvDataByte(), sendEvStatusMsg()\}$, i.e., $|O_1^{App}|$=6 (Figure 4.8). All the operations from the *PreDecoder* class ($M_1^{App}$) are copied to the *PreDecoder_proxy* class ($M_1^{Test}$) by the test framework generation procedure (algorithm 1).

However, for the operations $\forall O_i \in M_i \exists O_i^{Test}$ and $O_{if}^{Test} \neq 0$ is true, i.e., for every operation in the design model, there exists a corresponding operation/function call in the proxy test model. But the operation function body, in the proxy test model, is empty. This implies that though the proxy test model is created based on the design model, it derives only the abstract structure of the design model and not its functionality (line:6).

Therefore while adding the set of operations in the *PreDecoder* class ($M_1^{App}$) in the system design model, to the *PreDecoder_proxy* class ($M_1^{Test}$) in the test framework, the operation body, i.e., its implementation is not copied to the proxy class. In other words, the operations from the *PreDecoder* class ($M_1^{App}$) are copied to the *PreDecoder_proxy* class ($M_1^{Test}$). However, the implementation for the operations are not copied to the proxy class. For example, an operation *sendEvDataByte(int aByte)* in the *PreDecoder* class (Figure 4.8) comprises of a function body, (e.g. implementation for sending a message/byte). In the corresponding proxy class (i.e., *PreDecoder_proxy*) the implementation/function body (Figure 4.14) is empty.

– The next step of the algorithm (line:7-9) deals with copying the attributes ($A_n^{App}$), associations ($Q_{n^{App}}$) and dependencies ($D_n^{App}$) of the module $M_n^{App}$, to its respective proxy module $M_n^{Test}$ (newly created in the previous steps) in the proxy test model.

  Thus, for the *PreDecoder* class ($M_1^{App}$), the attributes $A_1^{App}$, associations $Q_1^{App}$ and dependencies $D_1^{App}$ are copied from the *PreDecoder* class to $A_1^{Test}$ (attributes), $Q_1^{Test}$ (associations) and $D_1^{Test}$ (dependencies) in the *PreDecoder_proxy* class (Figure 4.8, Figure 4.11). Here, $A_1^{App}$={*EventDataByte, EventStatusMsg, MIDICmd, ndxDB, ndxSM*}, $Q_1^{App}$={*itsMIDI_Interpreter*} and $D_1^{App} = \{\ll usage \gg UART0\}$. These elements are copied from the *PreDecoder* class ($M_1^{App}$) to its respective proxy module *PreDecoder_proxy* class in the test framework.

– If the processed module $M_n^{App}$ in the system design model is the chosen SUT, then the *createSUTStateChart()* procedure is invoked (line:10-11). The input parameters for *createSUTStateChart()* are the chosen SUT $M_{sut}^{App}$ and the state chart of SUT in design model $S_n^{App(s,t,e)}$ (line:11). The function *createSUTStateChart()* returns the state chart of the SUT in the proxy test model, $S_n^{Test(s'_{def},t',e')}$. The transitions of the state chart of the SUT in the proxy test model consist of functionality to communicate with the target debugger to insert the corresponding test data to the target system. A detailed discussion of *createSUTStateChart()* procedure is provided in algorithm 2.

  For the chosen SUT, the *PreDecoder* class ($M_1^{App}$) and the state chart of the SUT in the design model ($S_1^{App} = \{s, t, e\}$) are the input arguments for the *createSUTStateChart()* procedure. The state chart ($S_1^{App(s,t,e)}$) of the *PreDecoder* class ($M_1^{App}$) is shown in Figure 4.6. In this example, the set of states is represented as $s$={*StartSequence, NoteOnOff, OtherCommand, CMD_Received, DataByte1, DataByte2, ConsumeBytes*}, i.e., $|s|$=7. Similarly, the set of events is represented as $e$={*evStatusMsg(MIDI_Cmd), evDataByte(Byte)*}, i.e., $|e|$=2. In this example, the total number of transitions is represented as $|t|$=10. These are the input parameters for the *createSUTStateChart()* procedure.

- On the other hand, if the processed module in the system design model is not the chosen SUT (line:12) then an empty state chart is created in the respective proxy test module.

  For example, a state chart corresponding to the *MIDI_Interpreter* class is empty in its respective proxy class *MIDI_Interpreter_proxy*. This is because the *MIDI_Interpreter* class is not the chosen SUT.

- In the next step (line:15-17), the procedure *generateEventConsumedOnTarget*($M_n^{App}$) is invoked. This procedure creates the function *generateEventConsumed()* for all the proxy classes created for the chosen SUT in the proxy test model. The operation *generateEventConsumed()* is used to generate the respective event consumed on the target (on the host computer) while interpreting the test results in the proxy test model. Thus, the functionality in the operation *generateEventConsumed()* helps in mirroring the test case execution process (occurring in the target) on the host computer. This function is therefore invoked while interpreting the test results (form the target) in the test framework. A detailed discussion on the steps involved for creating the *generateEventConsumed()* operation for all the proxy classes, is provided in section 4.4.2.

3. In the next step (line:19), the UTP artifact $T_{Arch}$ in the test framework $P^{Test}$ is created by invoking the function *createUTPArtifact()*. The UTP artifact is based on the UTP [111] and defined as a tuple of six elements, namely $T_{Arch} = \{M_{sut}^{Test}, T_{comp}, T_{context}, T_{config}, T_{control}, T_{tc}\}$. The input parameters for this function are design model $P^{App}$=$M^{App}$, the test framework created so far $P^{Test}$, and the chosen SUT $M_{sut}^{App}$. A detailed description is provided in algorithm 3 (section 4.4.4). If the UTP artifact creation algorithm returns an error (line:20-22 in algorithm 1), then the algorithm skips further processing and returns an error message (line:20-22, 30).

4. The communication interface component ($T^{Com}$) is created in the next step for the chosen SUT (line:23) by the *createComObject()* procedure. $T^{Com}$ comprises of functionality to send, receive and decode the messages sent between the test framework and the target debugger. Hence, $T^{Com}$ is defined as a tuple, $T^{Com} = \{T_{send}^{Com}, T_{receive}^{Com}, T_{decode}^{Com}\}$. In the prototype, the communication interface component is implemented as a TCP/IP communication interface. A detailed description of the procedure to create the communication interface component of the test framework is provided in algorithm 4 (section 4.4.5). If the procedure to create the communication artifact component returns an error (line 24-26 in algorithm 1), then the algorithm skips further processing and returns an error message (line:30).

Once the algorithm generates the test framework and its components, $P^{Test} = \{M^{Test}, T_{Arch}, T^{Com}\}$, the algorithm terminates and returns (line:28). The two sub-procedures invoked during the proxy test model creation, i.e., *generateEventConsumedOnTarget*($M_n^{App}$) (invoked in line:16 in algorithm 1) and $createSUTStateChart(M_{sut}^{App}, S_n^{App(s,t,e)})$ (invoked in line:11 in algorithm 1) are discussed in sections 4.4.2 and 4.4.3 respectively. The steps involved in generating the UTP artifact component (algorithm 3) and the communication interface component (algorithm 4) for the test framework are discussed in sections 4.4.4 and 4.4.5 respectively.

### 4.4.2 Description of the *generateEventConsumedOnTarget* procedure

The main idea in the test framework generation algorithm is to generate the necessary artifacts (challenge (2)) on the host computer (i.e., the test framework) to support the model-based test case execution in embedded systems. In this context, the procedure *generateEventConsumedOnTarget()* is invoked during the test framework generation algorithm to create the function *generateEventConsumed()* for all the proxy classes in the proxy test model. The key functionality of this operation *generateEventConsumed()* in each proxy class is to generate the respective event consumed on target (at the host computer) while interpreting the test results in the proxy test model. Thus, it helps in mirroring the test case execution process (occurring in the target) on the host computer. This function is therefore invoked while interpreting the test results (from the target) in the test framework.

To gain further understanding, consider the following example. Consider that an event *evValidMIDICmd(MIDI_Cmd)* is generated from the *PreDecoder* class ($M_1^{App}$) to the *MIDI_Interpreter* class ($M_2^{App}$), as seen in Figure 4.15 in the target. The corresponding trace data for the above action is

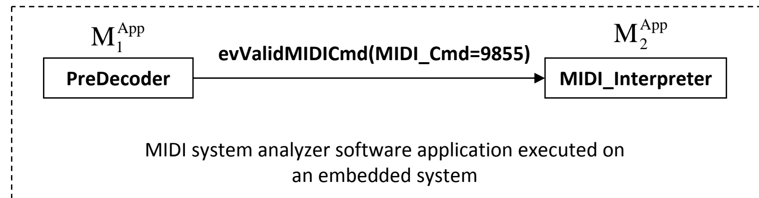



Figure 4.15: An event *evValidMIDICmd(MIDI_Cmd=9855)* is generated (on the target) from the *PreDecoder* class to the *MIDI_Interpreter* class

sent by the runtime monitoring routine to the target debugger in the pre-defined frame format (Figure 3.6). The series of steps involved in sending the trace data from the target to the test framework is shown in Figure 4.16. Consider the steps 1, 2 and 3 in Figure 4.16.

#### Steps 1, 2 and 3 in Figure 4.16

The target debugger receives the trace data in the pre-defined format (Figure 3.6) from the runtime monitoring routine in the embedded system. This is seen in steps 1 and 2 in Figure 4.16. The trace data received from the embedded system is communicated to the communication interface component in the test framework (step 3). The test framework is generated for a chosen SUT (i.e., the *PreDecoder* class in this example) at the host computer (Figure 4.16).

The test result, received at the target debugger, comprises of the event (e.g. sent as a 4 byte value), its source (4 bytes), destination (4 bytes), current time of execution on the target (4 bytes), current state (1 byte) of the destination object and the parameters for the event (if any). For the above example, the parameters such as event (0x00000673), source (0x0000183f), destination (0x00000db3) and current state (0x01) of the destination are sent to the host computer as seen in Figure 4.16 (steps 1 and 2). The current time (0x00102379) and the event parameters (0x0000267f) are also sent as hexadecimal values to the host (Figure 4.16). The event *evValidMIDICmd()* comprises of one integer parameter, namely *MIDI_Cmd* (sent as 4 bytes in test result).

The test result (comprising of 21 bytes of data) is mapped to their respective high-level (i.e., design-level) names (i.e., event, source, destination, state name) at the host computer by the target

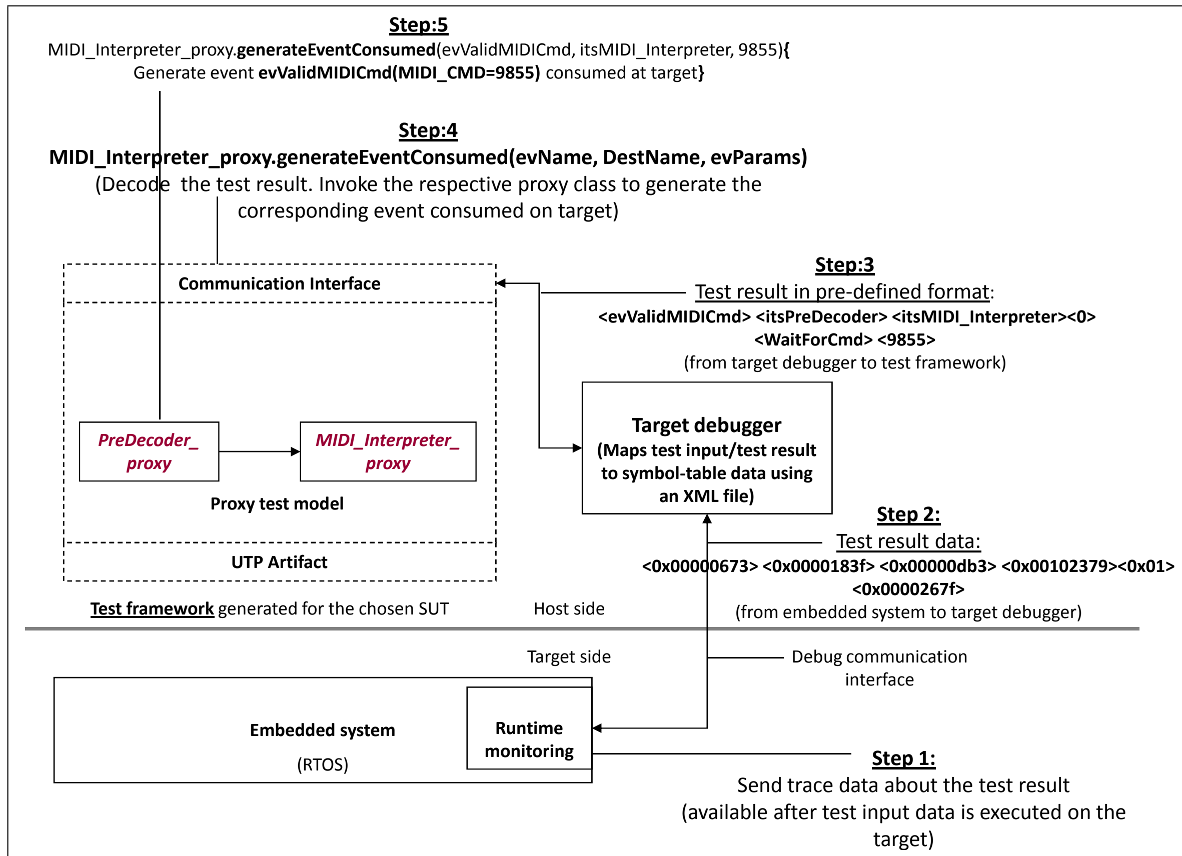



Figure 4.16: Trace data corresponding to an event execution, i.e., *evValidMIDICmd(MIDI_Cmd=9855)* generated on the target from the *PreDecoder* class to the *MIDI_Interpreter* class

debugger (step 2 in Figure 4.16). For the above example, the test result obtained, after mapping the received result to their high-level names by the target debugger, is $< evValidMIDICmd >$ $< itsPreDecoder > < itsMIDI_Interpreter > < 0 > < WaitForCmd > < 9855 >$.

In this example, the time value in the test result obtained from the target debugger is $< 0 >$ as this value denotes the time elapsed between the current event and the previous (immediate) event. This is useful than current time in the test results. For instance, in the RTOS framework used in the prototype, the timer granularity is set to 1 ms. Therefore, if time elapsed between two successive events is 0 ms, then it indicates that the successive events have occurred on the target in less than 1 ms time interval. The parameter value for the event *evValidMIDICmd*, i.e., in this example 0x0000267f is mapped to its decimal equivalent (9855), by the target debugger. After the steps 1, 2 and 3 (shown in Figure 4.16) are completed, the test result is available at the test framework.

### Steps 4 and 5 in Figure 4.16

Once the test results are available at the test framework (received from the target debugger after step 3 in Figure 4.16), the functionality of the test framework is to decode and interpret the test results. The test results are interpreted at the test framework (e.g. an event *evValidMIDICmd()* is sent from *PreDecoder* class to *MIDI_Interpreter* class and consumed by the *MIDI_Interpreter* class). This is carried out in steps 4 and 5 in Figure 4.15. Then the respective event is generated between the cor-

responding proxy classes in the test framework. For the above example, an event *evValidMIDICmd()* is generated from *PreDecoder_proxy* class to *MIDI_Interpreter_proxy* class in the test framework (host computer). This event is consumed at the *MIDI_Interpreter_proxy* class. This step (step 5 in Figure 4.16) helps in mirroring the test case execution process (at the target) on the host computer.

The steps 4 and 5 in Figure 4.16 illustrate the steps involved in decoding the test results (obtained from the target debugger) in the test framework. As seen in Figure 4.16, the test result obtained from the target debugger is received and processed by the communication interface component in the test framework (details in section 4.4.5). In short, it decodes the incoming test result from the target debugger and invokes the respective proxy class in the test framework.

Thus, based on decoding the incoming test result, the communication interface component invokes the *generateEventConsumed()* operation in the *MIDI_Interpreter_proxy* class. This is because, in the test result obtained (Figure 4.16), the *PreDecoder* class is the source for the event (*evValidMIDICmd*) generated to the *MIDI_Interpreter* class and the event is consumed by the *MIDI_Interpreter* class. This is shown as *MIDI_Interpreter_proxy.generateEventConsumed(evName, DestName, evParams)* in (step 4) Figure 4.16. Here, *evName=evValidMIDICmd* (event name), *DestName=MIDI_Interpreter* (destination name) and *evParams=9855* (event parameters) for the above example.

In step 5 (Figure 4.16), an event corresponding to the test result obtained from the target is generated on the host computer, by the *generateEventConsumed()* operation in the respective proxy class. Once the source, destination and event values are identified, the *MIDI_Interpreter_proxy.generateEventConsumed()* function generates the respective event on the host computer (Figure 4.16).

This functionality in the proxy class (i.e., the *generateEventConsumed()* operation) is automatically generated by the *generateEventConsumedOnTarget* procedure invoked during line:16 of algorithm 1. The steps involved in creating the *generateEventConsumed()* operation in each proxy class, by the *generateEventConsumedOnTarget*($M_n^{App}$) procedure is discussed below.

### The *generateEventConsumedOnTarget*($M_n^{App}$) procedure

One main goal for generating the test framework (challenge (2)) at the host computer is to mirror the test case execution process (on the target) at the host. In order to achieve this goal, each proxy class generated in the test framework comprises of an operation named as *generateEventConsumed()*. The key functionality of this operation is to generate an event on the host computer corresponding to the event that has occurred on the target. This action is carried out at the test framework by invoking the respective *generateEventConsumed()* operation in a corresponding proxy class, based on the decoded test results obtained from the target (Figure 4.16). The procedure *generateEventConsumedOnTarget*($M_n^{App}$) invoked in line:16 of algorithm 1, is used to generate the *generateEventConsumed()* operation for each proxy class. The series of steps involved in this procedure, invoked during the test framework creation, is shown in Figure 4.17.

Note that in the prototype developed in this thesis, the test framework generation algorithm is implemented in the programming language Java, with the aid of APIs [88] from a MBT tool, Rhapsody-Test conductor add-on [43]. Whereas, the components of the test framework such as proxy test model, UTP artifact are generated as UML elements. The operations/functions (e.g. TCP/IP communication interface, *generateEventConsumed()*) in the test framework are implemented in the programming language C. In the following, the *generateEventConsumedOnTarget()* procedure is explained with examples for a chosen class, namely, *MIDI_Interpreter* from the system design model.

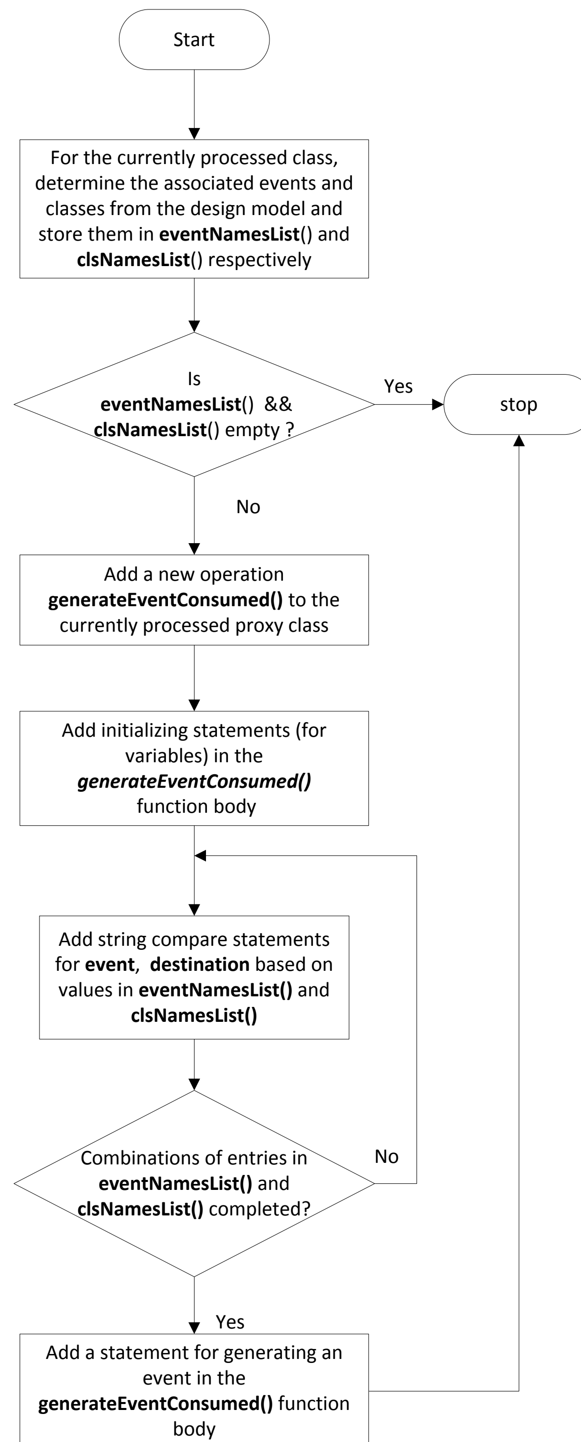



Figure 4.17: Series of steps involved in the *generateEventConsumedOnTarget*($M_n^{App}$) procedure in the test framework generation algorithm

In the first step (Figure 4.17), for the currently processed class ($M_n^{App}$), the associated classes and events are stored in data structures such as *eventNamesList()* and *clsNamesList()* (e.g. using *ArrayList* in Java). For example, in the *MIDI_Interpreter* class, *evNamesList*={*evValidMIDICmd*}

and *clsNamesList*={*MIDI_Interpreter*}.

In the next step, the procedure checks if the data structures containing the event names and the class names associated with the currently processed class (i.e., *eventNamesList()* and *clsNamesList()*) are empty. If it is empty, then the procedure stops further processing and returns. If it is not empty, the procedure performs the next step. In our example (Figure 4.15), *evNamesList()* and *clsNamesList()* are non-empty. Hence, the next step of the procedure, namely adding an operation *generateEventConsumed()* to the corresponding proxy class is carried out.

Initializing statements (e.g. for variables) are added by the procedure for the *generateEventConsumed()* function body, in the next step (Figure 4.17). These statements are required to declare and initialize the variables that are intended to be used in the next steps of this operation (*generateEventConsumed()*). An example of the initializer statements added in the *generateEventConsumed()* operation in the *MIDI_Interpreter_proxy* class is shown in lines:3-6 of the code snippet/sample below.

```
1  MIDI_Interpreter_proxy.generateEventConsumed(char* eventName, char* destName, char*
       eventParams){
2
3  int paramValue=null; /*To store parameter values for event, if any*/
4  struct RiCEvent_t *event; /*Data structure for an event*/
5  struct RiCReactive_t* destination; /*Data structure for a class*/
6  char *evName=eventName, *DestName=destName, *evParams=eventParams;
7
8  if ((strcmp(evName,"evValidMIDICmd")==0)){/*Check the name of the event*/
9    if ((strcmp(DestName,"itsMIDI_Interpreter")==0)){/*Check the destination of the
         event*/
10    /*Generate corresponding event consumed on host*/
11    destination = &(me->ric_reactive); /*Destination of the event*/
12    paramValue = atoi(evParams); /*Parameter for the event*/
13    event = (struct RiCEvent_t*)(RiC_Create_evValidMIDICmd(paramValue)); /*Create event
          */
14   }
15  }
16 /*Generate event on host based on decoded test result*/
17 (void) RiCReactive_gen(destination, event, RiCFALSE);
18
19 }/*end of MIDI_Interpreter_proxy.generateEventConsumed()*/
```

In the next step, statements to compare the event and destination names are added by the *generateEventConsumedOnTarget*($M_n^{App}$) procedure shown in Figure 4.17. The event and destination (i.e., class) names are obtained from the data structures *evNamesList()* and *clsNamesList()* respectively. The corresponding string compare statements added for the list of event and class names for the *MIDI_Interpreter_proxy* class are as shown in lines:8-15 in the code snippet. Consider the example shown in Figure 4.15 and 4.16. In this example, an event *evValidMIDICmd()* is generated from the *PreDecoder* class to the *MIDI_Interpreter* class. This event is consumed at the *MIDI_Interpreter_proxy* class. In order to interpret the test result corresponding to the above example (steps 3, 4 and 5 in Figure 4.16), the respective string comparison statements in the (*MIDI_Interpreter_proxy* class) is available in lines:8-17 (in the code snippet). In line:8, it is checked if the incoming test result comprises of an event *evValidMIDICmd*. In line:9, it is determined if the destination of the event is *MIDI_Interpreter* class. Once these two comparisons are complete, it is ascertained that an event

*evValidMIDICmd* has been generated from *PreDecoder* to *MIDI_Interpreter* class on target and the event *evValidMIDICmd* is consumed at the *MIDI_Interpreter_proxy* class. The statements in lines:11-13 prepare for generating the corresponding event (*evValidMIDICmd*) between the respective proxy classes (i.e., *PreDecoder_proxy* and *MIDI_Interpreter_proxy*) at the host computer. At this juncture, the event and destination values are determined corresponding to the test result. Thus, an event is ready to be generated and consumed at the host computer.

In the last step of the *generateEventConsumedOnTarget*($M_n^{App}$) procedure (Figure 4.17), a statement to generate the event, thus decoded, is added to the *generateEventConsumed()* operation body. For the above example, an event corresponding to the decoded test result is generated in line:17. The parameters for this function comprise of the event and the corresponding destination of the event. Thus, for the above example an event *evValidMIDICmd* is generated and consumed at the destination *MIDI_Interpreter_proxy* (line:17).

The statements generated for the *MIDI_Interpreter_proxy.generateEventConsumed()* operation in the above example are adapted to comply with the rules for code generation in the MDD tool Rhapsody [43], in which the prototype is evaluated. For example, the statement (lines:13) *event = (struct RiCEvent_t*)(RiC_Create_evValidMIDICmd(paramValue))* corresponds to creating an event (*evValidMIDICmd*) as per the rules of the MDD tool. Similarly, the statement *(void) RiCReactive_gen(destination, event, RiCFALSE )* in line:17 correspond to generating an event in the MDD tool Rhapsody, used in the prototype evaluation.

Note that the destination of the event consumed on target is already ascertained while decoding the test result at the communication interface component of the test framework. This component in turn invokes the *generateEventConsumed()* operation in the respective proxy class. In this example, corresponding to the decoded test result, the *generateEventConsumed()* operation in the *MIDI_Interpreter_proxy* class, i.e., *MIDI_Interpreter_proxy.generateEventConsumed(evName, DestName, evParams)* is invoked. An elaborate discussion on the functionality of the communication interface component of the test framework is available in section 4.4.5.

From the above discussion it is clear that the procedure *generateEventConsumedOnTarget()* invoked in line:16 of algorithm 1 is used to generate the operation *generateEventConsumed()* for all the proxy classes. Therefore, one of the main goals of generating a test framework at the host computer (challenge (2)), i.e., mirroring the test case execution process (on the target) at the host is achieved using the *generateEventConsumed()* operation. On the other hand, this operation is in turn invoked (during runtime) based on the decoded test results (trace data) obtained from the target (by the communication interface component), as seen in Figure 4.16.

### 4.4.3 State chart creation for the SUT in test framework

In this section, an algorithm for creating the state chart of the SUT in the proxy test model is presented. As mentioned in the chapters 2 and 3, one of the goals of this thesis is to enable the automatic generation of infrastructure support to convey the test input data from the test framework to the embedded system. The test input data which is now encapsulated in a custom-defined format (Figure 3.5), is stored as action text for the transitions created in the state chart of the SUT in the proxy test model. The transitions comprise of functionality to convey the test input data to be used in the test cases in the test framework.

To realize the aforementioned goal, corresponding to the state chart of the SUT in the system design model, a state chart is created for the SUT in the proxy test model using the *createSUTStateChart()* procedure. The main functionality of this state chart is to convey the test data (e.g. test input) to the target debugger. The test data conveyed by this state chart to the target debugger is organized in a pre-defined frame format, illustrated in Figure 3.5 in chapter 3. The test data is sent to the target debugger via the communication interface component ($T_{send}^{Com}$) in the test framework. This procedure is invoked in line:11 of algorithm 1, to create the state chart of the SUT in the proxy test model.

#### 4.4.3.a Input for state chart creation procedure

An important point to note is that, this thesis proposes an integrated model-based approach and test automation by generating a test framework for executing the model-based test cases in embedded systems (challenges (2) and (3)). Automatic generation of probable test cases and the test input data to be used in the test cases for model-based testing is, by itself, an extensive research area. This aspect is outside the goals of this thesis work. Examples of research work to generate model-based test cases are available in [9], [39], [53], [57], [76] and also supported in MBT tools such as [42]. In practice, alternatives such as a separate test model generated based on the requirements specification or abstract test models, behavior models, etc, generated from the system design models are used as inputs for probable model-based test case/test data generation. These activities are not among the goals of this thesis.

However, this thesis aims to automate (as far as possible) the entire test framework generation and execution of model-based test cases in embedded systems. Hence, the *createSUTStateChart()* procedure, outlined in algorithm 2, discusses a methodology to generate the infrastructure support to convey the probable test input data to be used by the test cases in the test framework. The infrastructure support for conveying the test input data is created based on the state chart of the SUT in the given system design model. Thus, this algorithm illustrates the steps involved in organizing the test input data in a pre-defined format and demonstrates the infrastructure support to send the (probable) test input data to the target debugger.

As mentioned above, the input for this procedure is the system design model of the chosen SUT in the proposed approach (and prototype implementation). In practice, this procedure can be applied to an abstract test model or behavior model (e.g. which is based on the requirement specification or system design model) for test data generation and organization in a pre-defined frame format. The outlined procedure takes as input only the respective interfaces (such as events, source, destination) from the design model and generates the corresponding infrastructure support to convey the test input data. It does not inherit the functional aspects from the given design model.

The methodology discussed in this algorithm is closely based on UML state charts. In general, this methodology is especially applicable for state-rich systems and parallel-systems, where the individual components are capable of being modeled by finite state machines and state charts.

#### 4.4.3.b Description of the *createSUTStateChart()* procedure

The input parameters for the algorithm 2 are the chosen SUT ($M_n^{App}$) and the state chart of SUT in the design model ($S_n^{App(s,t,e)}$). In the MIDI system analyzer example, the corresponding input parameters for this algorithm are the chosen SUT, i.e., *PreDecoder* class ($M_1^{App}$) and state chart of

*PreDecoder* class ($S_1^{App(s,t,e)}$), as shown in Figure 4.6. The output of this algorithm is the state chart of SUT in the proxy test model $S_n^{Test(s'_{def},t',e')}$. For the MIDI example, this is the state chart of *PreDecoder_proxy* class ($M_1^{Test}$) represented as $S_1^{Test(s'_{def},t',e')}$.

The state chart of the SUT in the proxy test model, automatically generated by algorithm 2 is shown in Figure 4.18. Examples from the MIDI system analyzer case study, i.e., the state chart of the *PreDecoder* class in the system design model (Figure 4.6) and state chart of SUT in the proxy test model (Figure 4.18) are cited during the description of each step of the algorithm.

**Algorithm 2** : State chart creation for the SUT in the test framework

**Input:** Chosen SUT $M_{sut}^{App}$ and state chart of SUT in design model $S_n^{App(s,t,e)}$
**Output:** State chart of SUT in proxy test model $S_n^{Test(s'_{def},t',e')}$

1: **if** $S_n^{App(s,t,e)} \neq 0$ **then**
2: Create empty state chart $S_n^{Test(s,t,e)}$ in $M_n^{Test}$. Add a default state $s'_{def}$ and default transition $t'_{def}$ in $S_n^{Test(s,t,e)}$
3: **for** $\forall e_m \in S_n^{App}$ **do**
4: ***trigger***={event name} ;assign event name
5: ***eventParam***={event parameters} ;assign event parameters
6: *action*={action text} ;assign action text in transition
7: construct string *actionTextTestData* ($Str_{t_l}^{Test}$) = ***createTestDataString***(trigger, eventParam, action)
8: add a new transition $t'_l$ in $s'_{def}$ with parameters={***trigger***, ***eventParam***, $Str_{t_l}^{Test}$}
9: **end for**
10: **return** $S_n^{Test(s'_{def},t',e')}$ ;state chart of SUT in proxy test model
11: **else**
12: **return** Error ;state chart of SUT in design model $S_n^{App(s,t,e)}$ is empty.
13: **end if**
14: **Sub-procedure** String ***createTestDataString***(trigger, guard, action)
15: event= trigger
16: event parameters= eventParam
17: **if** action$\neq$ null **then**
18: *source*={source of *action*}
19: *dest*={destination of *action*}
20: **return** string $T_{send}^{Com}(<event><source><dest><event\ parameters>)$
21: **else**
22: *source*={*TestEnv*}
23: *dest*={$M_n^{App}$} ;test data for self transition
24: **return** string $T_{send}^{Com}(<event><source><dest><event\ parameters>)$
25: **end if**

The first step of the algorithm 2 (line:1) checks if the state chart of the SUT in the design model is empty. If it is non-empty the algorithm proceeds with further processing (line:2-9). If the state chart of the SUT is empty, the algorithm returns with an error message (line:11-13). The state chart of the SUT, i.e., the *PreDecoder* class, is non-empty in Figure 4.6. Hence, the algorithm proceeds with the further processing (line:2-9).

In the next step (line:2), an empty state chart $S_n^{Test(s'_{def},t',e')}$ is created in the proxy test model of the SUT ($M_n^{Test}$) in the test framework. A default state $s'_{def}$ and default transition $t'_{def}$ are created in $S_n^{Test(s'_{def},t',e')}$. For example, the default state created in the state chart of the SUT in the proxy test model is named as *stateChartOfSUT_PreDecoder_proxy* in Figure 4.18. A default transition is also shown in Figure 4.18.

Next, the processing of each event in the state chart of the SUT in the design model is carried out

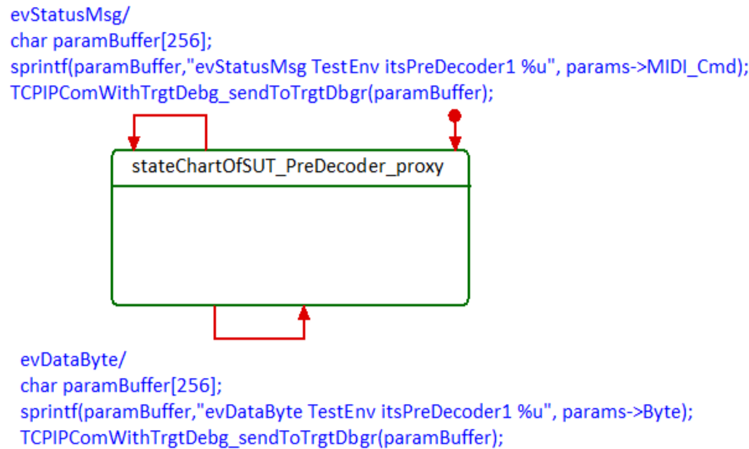



Figure 4.18: State chart generated automatically for the *PreDecoder_proxy* class by algorithm 2

(line:3). In the chosen SUT, the set of events $e$={*evStatusMsg(MIDI_Cmd), evDataByte(Byte)*}, i.e., $|e|$=2 is the input for this step.

For each event, the trigger or the event name, the event parameters and action text in transition are obtained in the next step. These values are stored in their respective data structures, which are string variables named *trigger*, *eventParam* and *action*. For example, for the transition with event *evStatusMsg(params→MIDI_Cmd=MIDI_NoteOn*||*MIDI_NoteOff)* shown in Figure 4.6, the corresponding data structures are *trigger*=evStatusMsg(), *eventParam*=(params→MIDI_Cmd) and *action text* is empty. An important point to note here is that, while the parameter name (*params→MIDI_Cmd*) is taken as input for the event *evStatusMsg()*, the parameter values (*MIDI_NoteOn*|*MIDI_NoteOff*) are not used by the algorithm. Thereby, a generic test input data is created by the algorithm without inheriting the functional behavior of the state charts in the system design model.

In the next step, the test input data corresponding to the currently processed event is created. This is performed by invoking the function *createTestDataString()* in line:7. The input parameters for this function are the string variables *trigger*, *eventParam* and *action*, obtained in the previous steps (line:4-6). The test data input created by this procedure and stored in a variable $Str^{Test}_{t_l}$ as seen in line:7. Thus, for the example chosen in the previous step, these values are *trigger=evStatusMsg()*, *eventParam=(params→MIDI_Cmd)* and *action=null*.

Now the required data for adding a new transition in the state chart of the SUT in the proxy test model is available. The parameters are trigger (i.e., event), event parameters and the action text for the transition created by *createTestDataString()* function. Hence, a new transition $t'_l$ is added in $s'_{def}$ in the state chart of the SUT where, parameters are {***trigger***, ***eventParam***, $Str^{Test}_{t_l}$}. This action is carried out in line:8 of algorithm 2. For the chosen example, a transition $t'_1$ is added in the state chart of the SUT *PreDecoder_proxy* in proxy test model ($M^{Test}_1$), where: ***trigger=evStatusMsg(),*** ***eventParam****=(params→MIDI_Cmd)* and ***Action text*** is shown in Figure 4.18.

The steps in lines:3 to 9 of the algorithm 2 are repeated for every event in the state chart of the SUT in the system design model. Once these steps are completed, the algorithm terminates. The output of the algorithm is the newly created state chart of the SUT in the proxy test model (e.g. Figure 4.18). From the description of the algorithm it is clear that for all the states in the state chart of the SUT in the system design model, there exists only one state in the state chart of the SUT in

the proxy test model. The transitions created in the state chart of the SUT comprise of test input data in their action text. This is automatically generated by the *createTestDataString()* procedure in algorithm 2.

#### 4.4.3.c Description of the *createTestDataString()* procedure

The main functionality of this procedure is to create the test data string to be sent to the target debugger using the communication interface component $T_{send}^{Com}$. In the prototype, $T_{send}^{Com}$ is named as *TCPIPComWithTrgtDebg_sendToTrgtDbgr()* (Figure 4.18).

The following describes the activities in the *createTestDataString()* procedure which is invoked for creating the action text corresponding to every event in the state chart of the SUT (line:7).

- In line:17, this procedure checks if the incoming action text is empty. If it is not empty (e.g. in case of events from one class to another), then the *source* and *destination* of the event are obtained from the action text. The test data string is then constructed in the pre-defined frame format (Figure 3.5).

  For instance, there are no transitions in the state chart of SUT in the system design model, from the *PreDecoder* class to another class (as seen in Figure 4.6). Hence an example of this case is not provided here.

- If the incoming action text is empty (line:21-24), then the test *source* of the test input data is set to *TestEnv*. This indicates that the test data is injected from the test environment (e.g. in case of self-transition in state chart of SUT in system design model). The *destination* is set as the SUT class itself.

  Both the events in state chart of the SUT (*PreDecoder* class) are used as self-transitions for the *PreDecoder* class. Hence, the test data created for *evStatusMsg(params→MIDI_Cmd)* is <*itsPreDecoder1 evStatusMsg TestEnv params→MIDI_Cmd*> and for *evDataByte(params→Byte)* is <*itsPreDecoder1 evDataByte TestEnv params→Byte*>. The various parameters of the test input data are space delimited. The above example is shown in the state chart of the SUT, automatically generated by this algorithm (Figure 4.6).
  The probable parameter values for the events, i.e., the values or ranges of *params→MIDI_Cmd* and *params→Byte* in the events *evStatusMsg()* and *evDataByte()* respectively, are not generated by the algorithm. This is because, these values are specified in the automatically generated or the manually specified test cases in the test framework. However, based on the values for these parameters in the test cases, the test data string is constructed dynamically during the test case execution. Hence, additional statements such as `char paramBuffer[256] and sprintf()` appear in the transitions of the state chart of the SUT (Figure 4.18) to accommodate the parameter values dynamically from the test cases.

Thus, this approach provides a generic algorithm to derive the infrastructure support for conveying the test input data, from state chart of the SUT in the system design model. The test data input can be used in the test cases in a test framework, such as the one described in section 4.4.1.

To gain further understanding, consider the series of steps involved in making use of the infrastructure support to convey the test input data (created automatically in the state chart of the SUT) in the proxy test model. Figure 4.19 shows the series of steps involved in sending the test input

data for a test case (specified in the test framework) to the embedded system, using the proposed approach. Consider that in an UML sequence diagram test case specified in the test framework (i.e.,

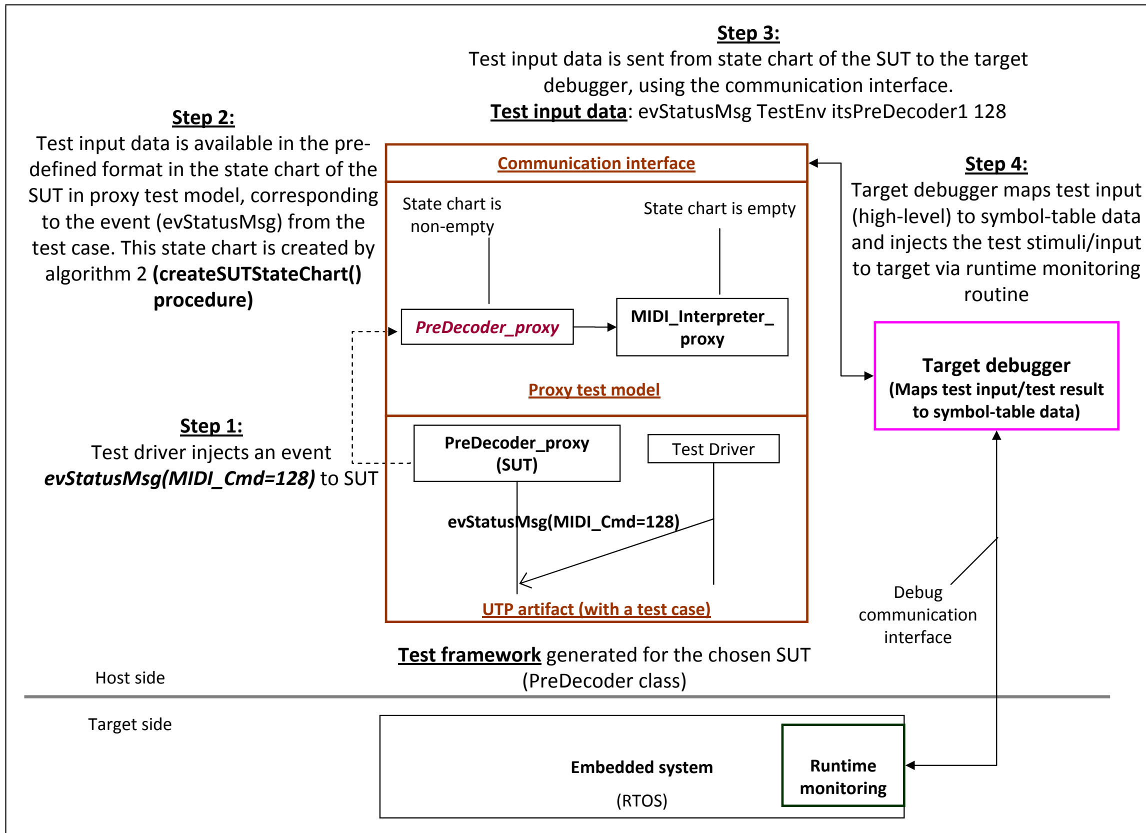



Figure 4.19: Series of steps involved in sending the test input data for a test case to the embedded system

the UTP artifact), a test driver injects an event *evStatusMsg(MIDI_Cmd=128)* to the SUT (i.e., the *PreDecoder_proxy*) class (step 1 in Figure 4.19). When an instance of a class (e.g. *PreDecoder_proxy*) is triggered by an event (e.g. *evStatusMsg(MIDI_Cmd=128)*, the state chart of the corresponding class reacts to the event. Note that the state chart of the SUT (i.e., the *PreDecoder* class) is already created by the algorithm 2 as shown in Figure 4.18. Therefore in the next step, (step 2 in Figure 4.19), the respective action for the transition with the event (e.g. *evStatusMsg(MIDI_Cmd=128)* is carried out. The corresponding action text statement in the state chart of the SUT (*PreDecoder_proxy* class) is shown in Figure 4.18. For the transition with the event *evStatusMsg(MIDI_Cmd=128)* in the state chart of the SUT (i.e., the *PreDecoder_proxy* class), the corresponding action text in the state chart of the SUT in the proxy test model (Figure 4.18) is shown below (in the programming language C).

```
1 sprintf(paramBuffer, "evStatusMsg TestEnv itsPreDecoder1 %u", params->MIDI_Cmd);
2 TCPIPComWithTrgtDebg_sendToTrgtDbgr(paramBuffer);
```

In the prototype implementation, the test framework generation algorithm is implemented in Java. The corresponding functionality (e.g. implementation of operations in the test framework), generated automatically by the prototype implementation of the algorithm is in the programming language C.

Here the test input data that is sent using the communication interface component is *"evStatusMsg*

*TestEnv itsPreDecoder1 128"* (line:1 in the above code sample). This is illustrated in step 3 in Figure 4.19. Thus, corresponding to the test case specified in the test framework (i.e., injecting an event from the external environment/test driver to the SUT), the test input data is already available in the pre-defined format in the state chart of the SUT in the proxy test model (i.e., in the state chart of the *PreDecoder_proxy* class).

In the next step (i.e., step 4 in Figure 4.19), the test input data received by the target debugger (from the test framework). The test data which is in high-level format (e.g. class names) is decoded and mapped to symbol-table data by the target debugger. The decoded test data is injected to the embedded system using the runtime monitoring routine in the target. Thus, from the above discussion, it is clear that the state chart of the SUT created by the algorithm 2 is used to provide infrastructure support to handle and convey the test input data in a pre-defined frame format. The test input data, in turn, corresponds to the injected event(s) in a given test case. The test cases are specified manually in the automatically generated test framework, in the proposed approach.

To summarize, the test framework proposed in this thesis, comprises of three main components namely, the *proxy test model*, *UTP artifact* and the *communication interface*. Among these three components of the test framework, the steps involved in generating all the elements in the *proxy test model* component of the test framework are discussed so far (in section 4.4.1, 4.4.2 and 4.4.3). In the following section (section 4.4.4), a detailed description of the procedure involved in the generation of the *UTP artifact* component of the test framework is provided. Similarly, in section 4.4.5, a detailed description of the steps involved in creating the *communication interface* component of the test framework is provided.

### 4.4.4 Procedure for UTP artifact generation

The UTP artifact component of the test framework, is envisioned to address the goals of this thesis pertaining to the usage of corresponding modeling languages for the MDD and the MBT phases in a generic test framework for embedded systems (challenges (1) and (2)). In this context, while UML is used for modeling and specification of the design model, UML and UTP artifacts are envisaged to be used for the test framework.

The UTP artifact component in the test framework provides structural/infrastructure support towards executing the model-based test cases in embedded systems. As the name implies, this component is created based on the artifacts provided in the UTP. Among the extensive set of stereotypes available in the UTP, elements from the *test architecture* concept group are used, in the UTP artifact component. A common methodology, which is used in this thesis, is to make use of an UML-based profile is to implement the profile as a set of stereotypes (in a MBT/MDD tool). Then the profile is included in a given project and specific elements of the profile (e.g. UTP) can be made use of (e.g. as a stereotype) for a chosen UML element.

A prototype of the test framework generation algorithm is implemented in the programming language Java, with the aid of the APIs [88] in the MDD/MBT tool Rhapsody [42], [43]. Thus, in the prototype, the UTP profile in the MDD/MBT tool is included in the generated test framework. The concise set of UTP elements used in this thesis are then generated, by including the required stereotypes from the UTP profile, in the UTP artifact creation algorithm (algorithm 3).

#### Input and output parameters for algorithm 3

The procedure for the UTP artifact generation (shown in algorithm 3) is invoked by the test framework generation algorithm (algorithm 1 in line:19). The data structures expected as input for this procedure are the system design model $P^{App} = M^{App}$, test framework $P^{Test}$ and chosen SUT $M_{sut}^{App}$.

For the chosen SUT, (*PreDecoder* class), the input parameters for this algorithm are the system design model (Figure 4.5), the test framework generated so far (comprising of the proxy test model-Figure 4.11) and the SUT (*PreDecoder* class).

The output from this procedure comprises of five elements represented as $T_{Arch}$, where $T_{Arch} = \{M_{sut}^{Test}, T_{comp}, T_{context}, T_{config}, T_{control}\}$. Note that the UTP artifact ($T_{Arch}$) generated by the UTP artifact creation algorithm comprises of only five elements (instead of six) as described in section 3.2.1.c and section 4.3. This is because, $T^{tc}$ is not generated by the approach proposed in this thesis, i.e., the model-based test cases are not automatically generated in the proposed approach. However, the model-based test cases are specified manually, in the automatically generated test framework. Therefore, the output of the UTP artifact generation algorithm represented by $T_{Arch}$ comprises of only five elements ($M_{sut}^{Test}$, $T_{comp}$, $T_{context}$, $T_{config}$, $T_{control}$) instead of six.

The procedure for UTP artifact generation shown in algorithm 3 is described below. The UTP artifact generated for the chosen SUT (*PreDecoder* class), using a prototype implementation of the algorithm, is shown in Figure 4.20

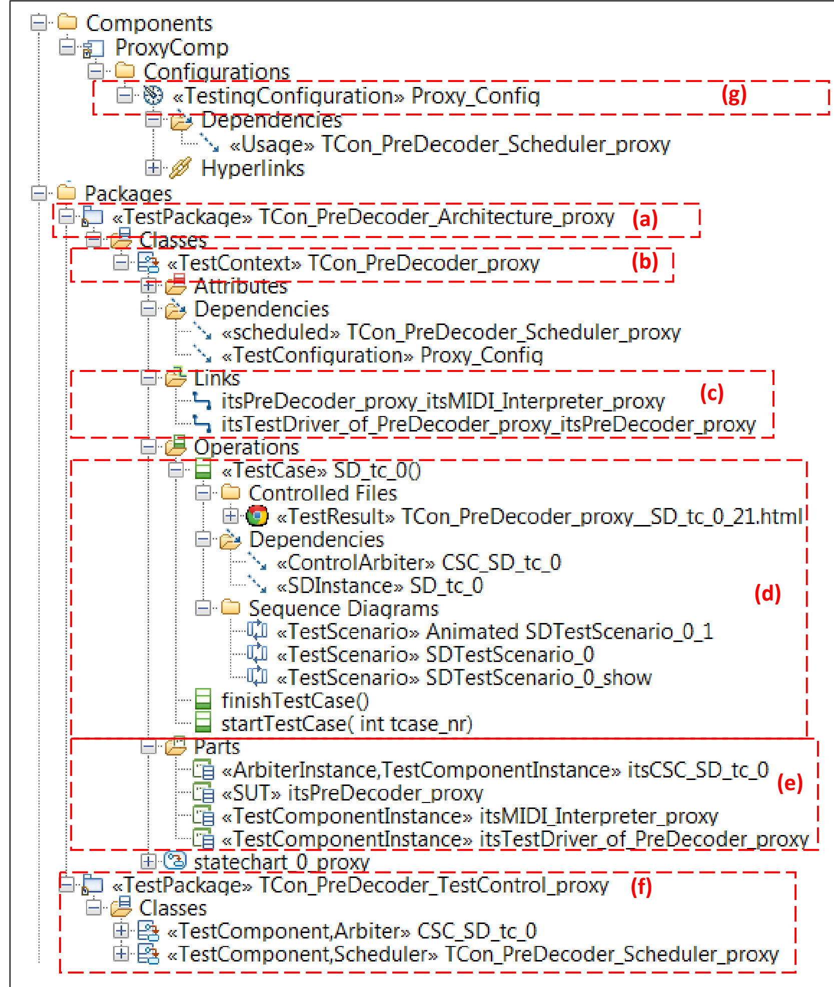
**Algorithm 3** : UTP artifact creation

**Input:** System design model $P^{App} = M^{App}$, test framework $P^{Test}$ and chosen SUT $M_{sut}^{App}$.
**Output:** UTP artifact component $T_{Arch} = \{M_{sut}^{Test}, T_{comp}, T_{context}, T_{config}, T_{control}\}$.

1: **if** $P^{Test} \neq 0$ **then**
2: Create a test package named *TCon_< $M_{sut}^{App}$ name>_Architecture_proxy* in $T_{Arch}$
3: Add a test context element $T_{context}$ named *TCon_¡$M_{sut}^{App}$ name>_proxy* in *TCon_< $M_{sut}^{App}$ name>_Architecture_proxy*
4: Create an instance of the SUT ($M_{sut}^{App}$ from the design model), referred as $M_{sut}^{Test}$ in $T_{context}$. Add a test configuration element $T_{config}$ for the test framework.
5: **for** $\forall M_n^{Test} \in P^{Test}$ **do**
6: create $T_{comp}$ in $T_{Arch}$ ;a test component instance is created for every proxy class in the test framework
7: **end for**
8: Add a test driver $T_{driver}$ in the created test component group $T_{comp}$
9: Create a test package $T_{control}$ named *TCon_< $M_{sut}^{App}$ name>_TestControl_proxy* in $T_{Arch}$ with *arbiter* and *scheduler* elements ;$T_{control}$ encompasses the *arbiter* and *scheduler* elements
10: **return** $T_{Arch} = \{M_{sut}^{Test}, T_{comp}, T_{context}, T_{config}, T_{control}\}$
11: **else**
12: **return** Error
13: exit
14: **end if**

#### Description of the UTP artifact generation procedure

In the first step of the algorithm 3, the test framework package created so far by the test framework generation algorithm is checked. If it is empty, then the UTP artifact creation procedure exits and reports an error (line:11-14). If the test framework created by the test framework generation algorithm




Legend: (a) Test package encompassing the UTP artifact component, (b) Test context, (c) Links, (d) Test case, (e) SUT, parts (f) Test control and (g) Test configuration

Figure 4.20: UTP artifact component created in the generated test framework for the *PreDecoder* class

(algorithm 1) is not empty (line:1 in algorithm 3), then the algorithm proceeds with the further processing (line:2-9). In the chosen example, the test framework package *TPkg_TCon_PreDecoder_proxy* (Figure 4.14) is non-empty. Hence, the algorithm proceeds with further processing (line:2-10).

In the next step (line:2), a test package element named $TCon_< M_{sut}^{App}\ name>_Architecture_proxy$ is added to $T_{Arch}$. For the chosen example, a test package element *TCon_PreDecoder_Architecture_proxy* is created as shown in Figure 4.20 (a).

A text context element $T_{context}$ named $TCon_<M_{sut}^{App}\ name>_proxy$ is added to the test package $TCon_< M_{sut}^{App}\ name>_Architecture_proxy$ (line:3) created in the previous step. For the chosen SUT, a test context element ($T_{context}$) named *TCon_PreDecoder_proxy* (Figure 4.20 (b)) is added to the test package *TCon_PreDecoder_Architecture_proxy.*

In line:4 of algorithm 3, an instance of the SUT $M_{sut}^{App}$ (based on the design model package) is created in $T_{context}$. This instance of the SUT is referred as $M_{sut}^{Test}$. Similarly, in this step, a test configuration element $T_{config}$ is created in the test framework. For the *PreDecoder* class, the UTP

artifact comprises of an instance of the SUT, *itsPreDecoder_proxy*, as seen in Figure 4.20 (e). For example, a test configuration element $\ll TestingConfiguration \gg$*Proxy_Config* is created as shown in Figure 4.20 (g).

In the next steps (line:5-7), a test component instance is created for every proxy class in the test framework. Thus, $\forall M_n^{Test} \in P^{Test}$ a test component instance $T_{comp}$ is created in the UTP artifact $T_{Arch}$. For the chosen example, the proxy classes are *MIDI_Interpreter_proxy* and *PreDecoder_proxy*. Therefore, an instance of the proxy class namely *MIDI_Interpreter_proxy* is created in this step. This is shown in Figure 4.20 (e), where the instance of the *MIDI_Interpreter_proxy* has a stereotype $\ll TestComponentInstance \gg$. Similarly, an instance of the proxy class, *PreDecoder_proxy* is also created at this step. However, since that is also the chosen SUT, the stereotype $\ll SUT \gg$ is applied for the instance of *PreDecoder_proxy* (Figure 4.20 (e)).

In line:8 a test driver $T_{driver}$ is added along with the test components created so far. The test driver is used for driving the test case execution process. For example, the proxy classes for the *PreDecoder* class in the test framework are *MIDI_Interpreter_proxy* and *PreDecoder_proxy*. Hence, an instance corresponding to these classes are created (as test components) in the UTP artifact. An instance of the test driver component namely, $itsTestDriver_of_PreDecoder_proxy$ is created for the chosen SUT as shown in Figure 4.20 (e). Note that the test component instances are grouped under a folder named "parts" in the prototype implementation.

The test control package element represented by $T_{control}$ is created in the next step (line:9) of the algorithm. The test control package is named *TCon_*< $M_{sut}^{App}$ *name*>*_TestControl_proxy* in $T_{Arch}$. For example, test control package element created for the *PreDecoder* class in the test framework is $TCon_PreDecoder_TestControl_proxy$ as shown in Figure 4.20 (f).

The $T_{control}$ component comprises of the *arbiter* and *scheduler* elements needed for controlling the test case execution process. The scheduler entity controls the running of the test cases. It keeps track of the creation and destruction of test components and gives instructions to the existing test components, when to start executing a given test case. The scheduler component for the chosen SUT (i.e., the *PreDecoder* class) in the generated UTP artifact is $\ll Scheduler \gg TCon_PreDecoder_Scheduler$-*_proxy* as shown in Figure 4.20 (f).

The *arbiter* component is a pre-defined interface provided with the UTP, which is used to determine the final verdict of a test case. This determination is performed based on an arbitration strategy and a default arbitration algorithm. In the prototype implementation, the default arbitration strategy and algorithm implemented in the MBT tool [43] is used. The test context uses the *arbiter* to evaluate the test results and assigns the overall verdict of a test case or a test context. The arbiter component created for the test context/test cases in the test framework package *TPkg_TCon_PreDecoder_proxy* is shown as $\ll arbiter \gg CSC_SD_tc_0$ in Figure 4.20 (f).

Once the algorithm generates the UTP artifact and its components, $T_{Arch} = \{M_{sut}^{Test}, T_{comp}, T_{context},$- $T_{config}, T_{control}\}$, the algorithm terminates and returns (line:9). The elements in the UTP artifact component for the chosen SUT is shown in Figure 4.20.

In order to execute the test cases, consisting of messages between several test components, there needs to be a valid link between the test components (tool specific). However, the links are not automatically generated in the prototype implementation of the algorithm. Therefore, in addition to the aforementioned automatically generated components, the end-user can manually specify the links between the test component instances (Figure 4.20(c)). For example, the link between the

instances of the *PreDecoder_proxy* class and the *MIDI_Interpreter_proxy* class specified manually is shown (*itsPreDecoder_proxy_itsMIDI_Interpreter_proxy*) in Figure 4.20(c).

As discussed in the beginning of this section, the automatically generated UTP artifact component comprises of $T_{Arch} = \{M_{sut}^{Test}, T_{comp}, T_{context}, T_{config}, T_{control}\}$. $T^{tc}$ is not automatically generated by the test framework generation algorithm, but specified manually (e.g. UML sequence diagram test case) by the end-user in the automatically generated test framework. For the chosen SUT, an UML sequence diagram test case is specified as seen in Figure 4.20 (d). The test case ($T^{tc}$) specified in this example is referred to as $\ll TestCase \gg$*SD_tc_0*. As per the rules of the MBT tool used in this prototype, the test case (e.g. $\ll TestCase \gg$*SD_tc_0*) is associated with a test scenario.

Thus, in this section the steps involved in the generation of the *UTP artifact* component were discussed. While UML is used for specifying the design model, UML and UTP based-elements are used in the automatically generated test framework. This is in line with the goals of challenges (1) and (2) pertaining to the use of corresponding modeling languages for the MDD and the MBT phases. The *UTP artifact* component in the test framework is used to provide infrastructure support for the test case execution process.

### 4.4.5 Communication interface creation

A generic approach towards generating a model-based test framework at the host computer, for automating the test case execution process during the MBT phase, is proposed in this thesis (challenge (2)). The test framework generation algorithm discussed in section 4.4.1 deals with the creation of a test framework with three components, namely *proxy test model*, *UTP artifact* and *communication interface*. The procedure to create the communication interface component is invoked in line:23 (*createComObject()*) of algorithm 1.

The communication interface component ($T^{Com}$) in the test framework is used by the test framework to send and receive the test data between the target debugger and the host computer. It also comprises of the functionality to decode, interpret and invoke the respective functions based on the received message (e.g. test results) from the target debugger. Therefore, this component in the test framework needs to be in its own thread of execution, in order to facilitate uninterrupted communication between the test framework and the target debugger.

The communication interface component of the test framework can be implemented using alternatives for inter process communication such as the COM [89] interface or as a TCP/IP interface (Figure 4.12). In the prototype, the *createComObject()* procedure creates a TCP/IP based communication interface, which makes use of the Winsock API [72]. The decision to choose the TCP/IP (sockets) interface as the alternative for implementing the communication interface (in the prototype) is based on the relative simplicity of implementing such an interface.

The functionality of the communication interface and its components created by the algorithm 4 is discussed in this section. In the prototype implementation, this interface is created as a singleton, active object. This means that there can be only one instance of the communication interface component running in the test framework. By being an active class object it indicates that when instantiated the class controls its own execution. Rather than being invoked or activated by other objects, it operates stand alone and defines its own thread of behavior. For example, in the MDD/MBT tool, Rhapsody (in which the prototype is evaluated) the functionality of the active class object is

implemented by overloading a function named *execute()*. Hence, in the generated test framework, the communication interface singleton active object overloads the *execute()* operation and implements the functionality for communicating with the target debugger. The tasks for the communication interface object include

- Sending messages to the target debugger (e.g. test data inputs)
- Receiving messages from the target debugger (e.g. test results from the embedded system)
- Decoding the received messages from the target debugger
- Invoking the respective functions in the test framework based on the decoded results in the previous step.

In the prototype, the test framework generation algorithm is implemented in Java, with the help of the APIs provided by the MDD tool Rhapsody [88]. Hence, the *createComObject()* procedure, is also implemented in a similar fashion. The output of the *createComObject()* procedure is the singleton, active object $T^{Com}$. In the prototype this is represented as *TCPIPComWithTrgtDebg()* operation. The algorithm for the *createComObject()* procedure is described below. Examples from the communication interface creation for the chosen SUT (*PreDecoder* class) in the MIDI system analyzer example are cited during the description of the various steps of the algorithm. However, note that this is a generic procedure for creating a TCP/IP based communication object for the test framework. It is independent of the embedded software application under consideration.

**Algorithm 4** : Communication interface creation

**Input:** System design model $P^{App} = M^{App}$, test framework $P^{Test}$ and chosen SUT $M^{App}_{sut}$ in the design model
**Output:** Communication interface component $T^{Com} = \{T^{Com}_{send}, T^{Com}_{receive}, T^{Com}_{decode}\}$

1: **if** $P^{Test} \neq 0$ **then**
2: Create an empty, singleton, active object $T^{Com}$ in $P^{Test}$ ;TCP/IP communication interface ;component
3: Add attributes required for TCP/IP based operations in $T^{Com}$ and assign initial values ;e.g. *clientSocket=0*, *clientSocketData=0*, *WSAData=0*
4: **for** $\forall M^{Test}_n \in P^{Test}$ **do**
5: Create ≪*Usage*≫ dependency on $M^{Test}_n$ in $T^{Com}$
6: **end for**
7: Create ≪*Usage*≫ dependency on $T_{context}$ element in $P^{Test}$
8: Add operations $T^{Com}_{send}$, $T^{Com}_{receive}$ and $T^{Com}_{decode}$ to $T^{Com}$ ;functionality to send, receive and decode ;messages between the test framework and target debugger
9: Add functionality/operation body for $T^{Com}_{send}$, $T^{Com}_{receive}$ and $T^{Com}_{decode}$ in $T^{Com}$
10: **return** $T^{Com} = \{T^{Com}_{send}, T^{Com}_{receive}, T^{Com}_{decode}\}$
11: **else**
12: **return** Error
13: exit
14: **end if**

#### 4.4.5.a Input and output parameters for algorithm 4

The *createComObject()* procedure described in algorithm 4 takes as input the given system design model $P^{App} = M^{App}$, the test framework $P^{Test}$ and the chosen SUT $M^{App}_{sut}$ in design model. For

the chosen SUT (*PreDecoder* class in MIDI system case study), the input parameters are the system design model (Figure 4.11), the test framework created so far comprising of the proxy test model (Figure 4.14), UTP artifact (Figure 4.20) and the chosen SUT (Figure 4.8).

In the prototype, this procedure is implemented as a TCP/IP communication interface [72] component. Thus, the output of this procedure is the TCP/IP communication interface denoted by $T^{Com} = \{T^{Com}_{send}, T^{Com}_{receive}, T^{Com}_{decode}\}$. The TCP/IP communication interface created (automatically), by the test framework generation algorithm, for the *PreDecoder* class is shown in Figure 4.21. The compo-

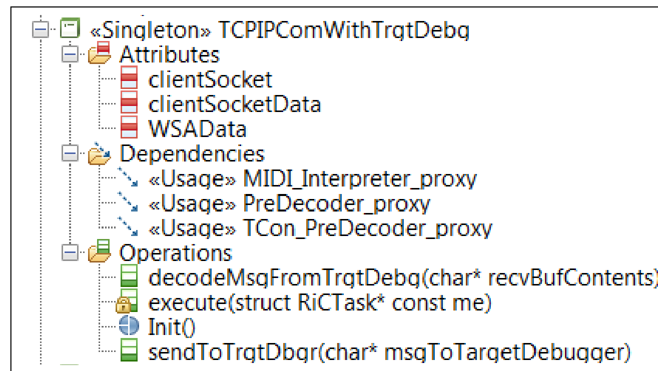



Figure 4.21: TCP/IP communication interface created for the chosen SUT (*PreDecoder* class)

nents of the communication interface namely, $T^{Com}_{send}$, $T^{Com}_{receive}$ and $T^{Com}_{decode}$ are represented as operations *sendToTrgtDbgr()*, *execute()* and *decode()* respectively (Figure 4.21) in the prototype implementation of the algorithm.

#### 4.4.5.b Description of *createComObject()* procedure

The first step of the algorithm checks if the test framework denoted by $P^{Test}$ is empty (line:1 in algorithm 4). If it is not empty then the algorithm proceeds with the further steps in creating the communication interface component (line:2-9). If the test framework is empty, it indicates an error in the initial steps (lines:2-15) of the algorithm 1. In this case, this procedure (algorithm 4) skips further processing and returns (line:10-13).

For the chosen SUT (*PreDecoder* class), the test framework (created so far by the algorithm 1) which is the input for the *createComObject()* procedure comprises of the proxy test model component (Figure 4.14) and the UTP artifact component (Figure 4.20). Hence, the algorithm 4 proceeds with further processing (line:2-9 in algorithm 4).

When the test framework package (Figure 4.14) is non-empty, in the next step (line:2) of the algorithm 4, an empty, singleton, active object $T^{Com}$ is created in $P^{Test}$. The attributes required for the TCP/IP based operations in $T^{Com}$ is created in the next step (line:3) of algorithm 4.

For the chosen example, a singleton, active object *≪Singleton≫ TCPIPComWithTrgtDebg* is created (Figure 4.21). As seen in Figure 4.21, attributes such as *clientSocket*, *clientSocketData* and *WSAData* are added to the communication interface component *≪Singleton≫ TCPIPComWithTrgtDebg*. These data structures are used to aid in establishing the TCP/IP communication interface functionality between the test framework and the target debugger.

The communication interface component *≪Singleton≫ TCPIPComWithTrgtDebg* includes the functionality to decode and invoke the respective operations in the test framework ($T^{Com}_{decode}$). This func-

tionality is used by the proxy classes created (already) in the proxy test model and the test cases specified in the UTP artifact component. Hence, in the next step (line:5) of the algorithm 4, the algorithm creates ≪*Usage*≫ dependencies on all the proxy classes created in line:5 of algorithm 1. In addition a ≪*Usage*≫ dependency is also created with the test context element ($T_{context}$) of the UTP artifact created by the test framework generation algorithm.

For example, the proxy classes created for the chosen SUT (*PreDecoder* class) in the test framework are *PreDecoder_proxy* and *MIDI_Interpreter_proxy*. Hence, a ≪*Usage*≫ dependency is created on these classes for the ≪*Singleton*≫*TCPIPComWithTrgtDebg* communication interface component, as shown in Figure 4.21. In addition, a ≪*Usage*≫ dependency is created on the test context element in the UTP artifact component of the chosen SUT, namely *TCon_PreDecoder_proxy*.

The communication interface component which has its own thread of execution, should be able to listen to the connected port in the target debugger. It should also be able to receive the messages from the target debugger continuously (while the test framework is active). This functionality is grouped in the component $T^{Com}_{receive}$ in $T^{Com}$. Similarly, this component is also responsible for sending and receiving messages between the target debugger and the test framework. This functionality is grouped in the components $T^{Com}_{send}$ and $T^{Com}_{decode}$ (in $T^{Com}$) respectively. Therefore, in line:8-9 of algorithm 4 the operations $T^{Com}_{send}$, $T^{Com}_{receive}$ and $T^{Com}_{decode}$ are added by the algorithm 4.

In the prototype, the aforementioned functionalities are implemented in *execute()* ($T^{Com}_{receive}$), *decodeMsgFromTrgtDebg()* ($T^{Com}_{decode}$) and *sendToTrgtDbgr()* ($T^{Com}_{send}$) operations in the communication interface component ≪*Singleton*≫*TCPIPComWithTrgtDebg* ($T^{Com}$) as seen in Figure 4.21. Note that, the functionality of ($T^{Com}_{receive}$) is implemented by overloading an operation named *execute()* in the prototype. This aspect is tool specific and corresponds to the working principle of the *active class* objects in the MDD tool [43]. Therefore, in the prototype, the communication interface functionality is implemented by overloading the *execute()* function in the ≪*Singleton*≫*TCPIPComWithTrgtDebg* component. The functionality of the $T^{Com}_{receive}$ (*execute()*), $T^{Com}_{decode}$ (*decodeMsgFromTrgtDebg()*) and $T^{Com}_{send}$ (*sendToTrgtDbgr()*) in $T^{Com}$ (≪*Singleton*≫*TCPIPComWithTrgtDebg*) are described in the following section.

#### 4.4.5.c Functionality of $T^{Com}_{receive}$ (*execute()*)

The functionality of the *execute()* operation is illustrated in the flowchart in Figure 4.22. As a first step, the *execute()* operation establishes a TCP/IP based connection between the test framework and the target debugger. An error handler routine handles any connection/communication errors. Once a connection is established successfully, the *execute()* operation acting as a stand alone thread listens on the TCP/IP port corresponding to the established connection. Once the trace data is received from the target debugger, it invokes the *decodeMsgFromTrgtDebg()* operation ($T^{Com}_{decode}$) with the received message as the parameter. In the prototype, the attributes such as *clientSocket*, *clientSocketData* and *WSAData* are added to the communication interface component ≪*Singleton*≫*TCPIPComWithTrgtDebg* (Figure 4.21), by the algorithm 4. These attributes are used by the *execute()* operation (in the prototype implementation) while establishing a TCP/IP connection between the test framework and the target debugger.

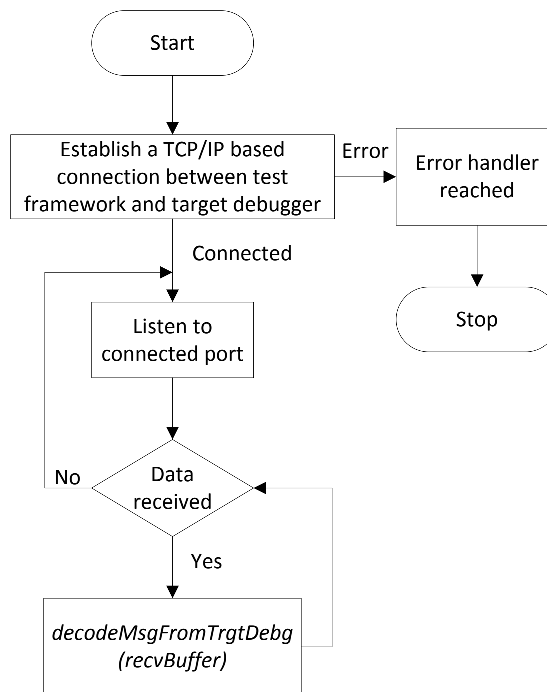



Figure 4.22: Functionality of the *execute()* operation in the TCP/IP communication interface

#### 4.4.5.d Functionality of $T^{Com}_{decode}$ (*decodeMsgFromTrgtDebg()*)

The *decodeMsgFromTrgtDebg()* operation in the ≪*Singleton*≫ *TCPIPComWithTrgtDebg* communication interface component of the test framework is responsible for decoding the received messages from the target debugger. The sequence of operations performed by the *decodeMsgFromTrgtDebg()* operation in the test framework is illustrated in Figure 4.23. This series of steps is described in section 4.4.5.e. The procedure to generate the decoding logic for the *decodeMsgFromTrgtDebg()* operation is handled by the test framework generation algorithm and implemented in the programming language Java. This procedure is illustrated in Figure 4.24 and described in section 4.4.5.f.

#### 4.4.5.e Series of steps in the *decodeMsgFromTrgtDebg()* operation

The sequence of steps performed by the *decodeMsgFromTrgtDebg()* operation in the test framework is illustrated in Figure 4.23. Upon interpreting the incoming messages from the target debugger, it is also responsible for invoking the respective operations in the proxy test model of the test framework. In the background, the actions are represented by the respective UML sequence diagrams (in the MDD/MBT tool), on invoking these operations. The messages received from the target debugger correspond to the actual behavior of the target, in real time. Therefore, by invoking the respective operations at the host computer, based on the decoded messages, the behavior of the target is reconstructed at the host and visualized using UML diagrams (e.g. sequence diagram) at the host computer.

To gain further understanding, consider the example shown in Figure 4.15. In this example, an event *evValidMIDICmd(MIDI_Cmd)* is generated from the *PreDecoder* class ($M_1^{App}$) to the *MIDI_Interpreter* class ($M_2^{App}$) (Figure 4.15) in the target. The corresponding trace data for the above action is

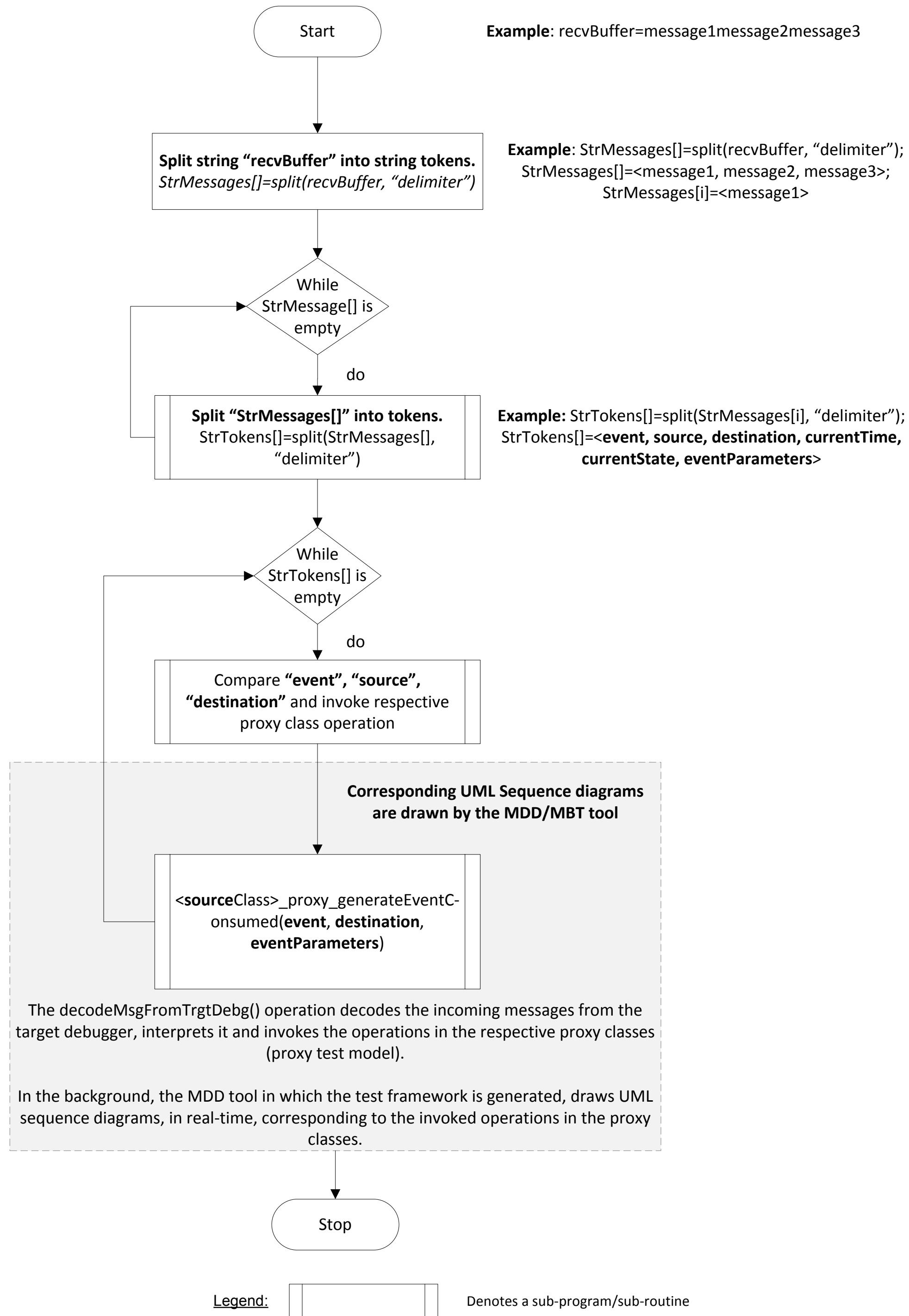



Figure 4.23: Functionality of *decodeMsgFromTrgtDebg()* in TCP/IP communication interface

sent by the runtime monitoring routine to the target debugger in the pre-defined frame format (Figure 3.6). The series of steps involved in sending the trace data from the target to the test framework is shown in Figure 4.16.

Consider the step 4 in Figure 4.16 in detail. In this step the test result (trace data) received from

the target is decoded and interpreted by the communication interface component. As mentioned in the previous section, the *execute()* operation ($T^{Com}_{receive}$) receives the incoming trace data. The trace data is then sent for decoding and further processing to the *decodeMsgFromTrgtDebg()* ($T^{Com}_{decode}$) operation in the communication interface component. The functionality of this operation is described using a flowchart in Figure 4.23. The received message (from $T^{Com}_{receive}$, i.e., the *execute()* operation) is available in a string buffer, *"recvBuffer"* in the prototype. The trace data received from the target may comprise of one or more test results, separated by a delimiter (Figure 4.23). Each message comprises of the test results in the pre-defined format (Figure 3.6). In the next step, each message comprising of an event, source, destination, current time, current state of the destination object and event parameters. These tokens correspond to the pre-defined format for sending the test results.

As seen in Figure 4.23, the set of string tokens corresponding to a test result is compared in the *decodeMsgFromTrgtDebg()* operation body. This is performed to determine the event, source and destination of the incoming trace data. For the example shown in Figure 4.16, the trace data for event consumed notification from the target debugger is $< evValidMIDICmd > < itsPreDecoder > < itsMIDI_Interpreter > < 0 > < WaitForCmd > < 9855 >$. The mapping of the components of the trace data to the pre-defined frame format is shown in Table 4.3.

Table 4.3: Mapping of the trace data frame format to an example (in Figure 4.16)

| **Parameters in trace data** | **Corresponding string tokens** |
|---|---|
| event | evValidMIDICmd |
| source | itsPreDecoder |
| destination | itsMIDI_Interpreter |
| time elapsed | 0 |
| current state | WaitForCmd |
| event parameters | 9855 |

In this example, the string tokens to be compared in the *decodeMsgFromTrgtDebg()* operation body, are the event $< evValidMIDICmd >$, the source $< itsPreDecoder >$ and the destination $< itsMIDI_Interpreter >$ . The code snippet/sample showing the respective decoding functionality for the above trace data (in the *MIDI_Interpreter_proxy.generateEventConsumed()* operation) is available below.

```
1  if ((strcmp(evName,"evValidMIDICmd")==0)){/*Check the name of the event*/
2      if ((strcmp(sourceName,"itsPreDecoder")==0)){/*Check the source*/
3          if ((strcmp(destName,"itsMIDI_Interpreter")==0)){/*Check the destination*/
4
5          /*Prepare the parameters for invoking the generateEventConsumed() operation*/
6          eventName="evStatusMsg";/*Assign the name of the event*/
7          destName="MIDI_Interpreter"; /*Destination of the event*/
8          eventParams=resultArray[5]; /*Parameters for the event from the test result*/
9
10         MIDI_Interpreter_proxy_generateEventConsumed(itsPreDecoder_proxy, eventName,
               destName, eventParams); /*Invoke the respective proxy class*/
11
12         }/*End of checking the destination name*/
13     }/*End of checking the source name*/
14 }/*End of checking the event name*/
```

As seen above, the string tokens are compared and the trace data (for the example shown in Figure 4.15) is interpreted in lines:1-3. Based on these comparisons, it is determined that an event *evValidMIDICmd* has been generated from the *PreDecoder* class to the *MIDI_Interpreter* class on the target. Therefore, the respective function in the test framework (i.e., the proxy class) is invoked to generate the event that has been consumed on the target. This is performed by invoking the *generateEventConsumed()* operation in the proxy class corresponding to the destination of the event. In this example (Figure 4.15), the event name, destination name and the event parameters are stored in variables *eventName*, *destName* and *eventParams* as seen in lines:6-8. Then, in the next step, the *generateEventConsumed()* operation in the proxy class (corresponding to the destination of the event, i.e., *MIDI_Interpreter_proxy* class) is invoked, as seen in line:10 in the code snippet.

Thus, the *decodeMsgFromTrgtDebg()* operation, decodes the incoming messages from the target debugger and invokes the operations in the respective proxy class (e.g. corresponding to the destination of the event). In the background, the MDD/MBT tool in which the test framework is generated (automatically), draws UML sequence diagrams, in real time corresponding to the invoked operations in the proxy classes (Figure 4.23).

#### 4.4.5.f Series of steps to generate the *decodeMsgFromTrgtDebg()* operation (function body)

The procedure to generate the decoding logic for the *decodeMsgFromTrgtDebg()* operation is handled by the test framework generation algorithm. This is implemented in the programming language Java using the APIs supported by the MDD tool [42]. This procedure is illustrated in Figure 4.24.

After adding variables and the functionality to process the incoming messages, this procedure adds the decoding logic for interpreting the messages processed so far (Figure 4.24). These messages are interpreted based on the event, source and destination values that is sent by the target debugger (i.e., the test result available in a pre-defined frame format shown in Figure 3.6). For a chosen SUT, the corresponding event and its respective classes are stored in a list by parsing the system design model. These values are stored in data structures *eventNamesList()* and *clsNamesList()* (e.g. `ArrayList` in Java) respectively. Based on these values, string compare statements are added to the function body of *decodeMsgFromTrgtDebg()*. These statements are added as C code by the test framework generation algorithm, for all possible combinations of the event, source and destination. These statements are used while decoding the trace data and determining the event, source and destination from the test result. Note that the test framework generation algorithm itself is implemented in the programming language Java with the aid of the APIs in the MDD tool Rhapsody in the prototype implementation. An example of the decoding logic, generated in the programming language C, by the test framework generation algorithm is available in the code snippet in section 4.4.5.e.

#### 4.4.5.g Functionality of $T^{Com}_{send}$ (*sendToTrgtDbgr()*)

The functionality of this operation is to send a given message to the target debugger. This is implemented using the *send* functionality available in the Winsock [72] API in the prototype. Errors in communication (e.g. while sending a message) are handled by an error handler. The *sendToTrgtDbgr()* ($T^{Com}_{send}$) operation is in turn invoked by the state chart of the SUT ($S_n^{Test(s,t,e)}$) in the proxy test model. This is used to send messages (e.g. test input data in a pre-defined format shown in

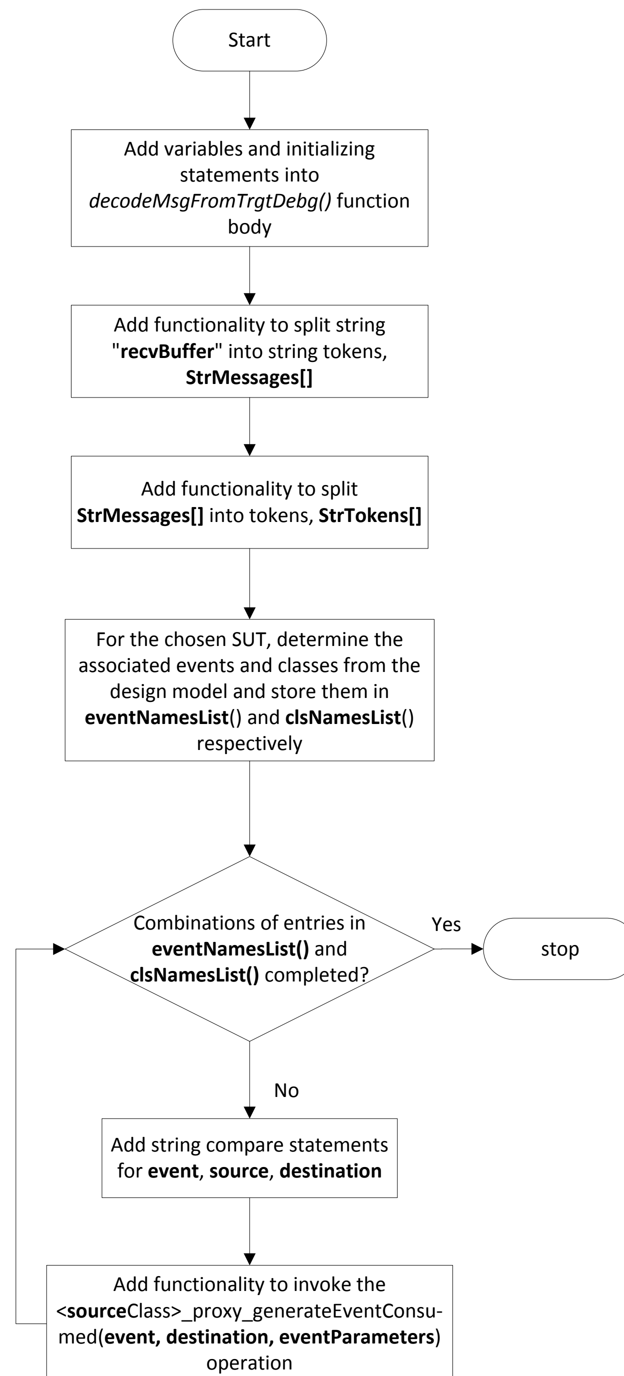



Figure 4.24: Steps involved in the *createComObject()* procedure to generate the decoding logic for *decodeMsgFromTrgtDebg()* function

Figure 3.5) to the target debugger. In the prototype implementation, this function is named as *TCPIPComWithTrgtDebg()* (Figure 4.18).

Summarizing, in section 4.4 a generic test framework generation algorithm has been introduced and explained in detail with illustrative examples. The sub-procedures invoked by this algorithm are

also discussed in detail with examples from the MIDI system analyzer case study (challenges (1) and (2)). In the following section (section 4.5), a complexity analysis of the test framework generation algorithm is provided.

## 4.5 Complexity Analysis

A complexity analysis for the proposed approach and the test framework generation algorithm (algorithm 1) can be provided in two perspectives. They are

- The test framework *generation* complexity.
- The *runtime* complexity of the test framework/proposed approach while executing the test cases in the embedded system.

The time and memory (size) complexity involved in both these perspectives are discussed below. The complexity analysis is provided on the basis of the RAM model of computation [95]. This section provides a theoretical analysis on the derivation of the complexity measures for the test framework generation algorithm. These theoretical estimates are recalled and compared with the experimental results, pertaining to the parameters in the complexity measures, in chapter 5.

### 4.5.1 Test framework generation complexity

This section provides an analysis on the complexity measures for the test framework *generation* by the test framework generation algorithm (algorithm 1) discussed in section 4.4.1. This analysis gives an important insight into the processing requirements of the algorithm for the test framework generation. An expression for time and memory complexity of the test framework generation algorithm can be defined as a sum of the complexities involved in generating its three components, namely the *proxy test model*, the *UTP artifact* and the *communication interface*. In order to calculate the generation complexity of this algorithm, the following execution parameters are taken into consideration (Figure 4.25).

- $\psi$ denotes the number of SUTs (e.g. a *class* element in design model).
- $a$ is the number of association ends of a given SUT.
- $c$ is the number of classes in each association end $a$. This can take a maximum value of two, i.e., maximum of two classes in an association end. This does not refer to the multiplicity of an association end.
- $e$ is the number of event receptions in a given class $c$.
- $s$ is the number of states in the state chart of the given SUT in the design model.
- $t$ is the number of transitions in the state $s$ in the state chart of the SUT.
- $\beta$, $\gamma$ and $\delta$ denote the respective processing complexities for the generation of the three components of the test framework namely, the *proxy test model*, *communication interface* and the *UTP artifact* respectively.

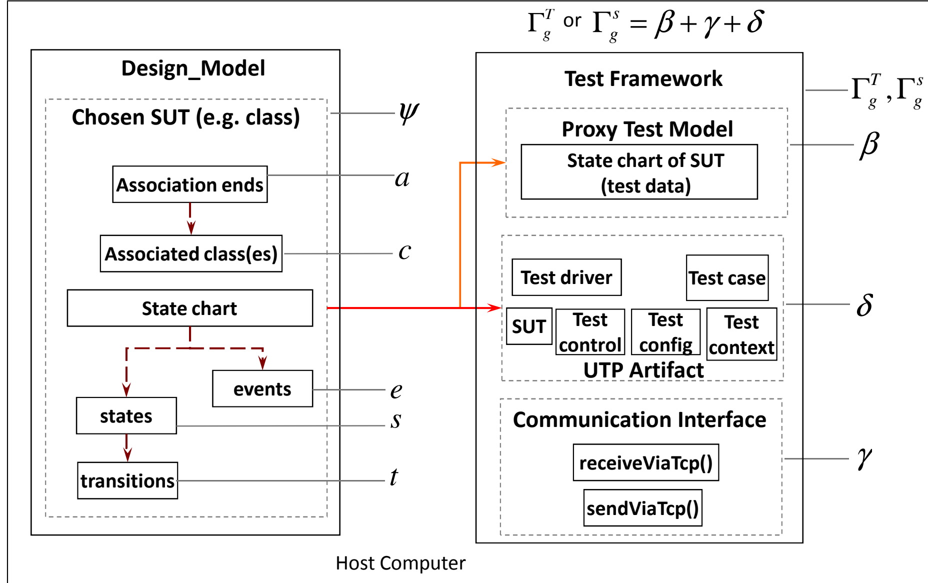



Figure 4.25: Execution parameters for test framework generation

- $\Gamma_g^T$ and $\Gamma_g^S$ denote the approximate time and memory-size *generation* complexities of the test framework generation algorithm (algorithm 1 in section 4.4.1).

The test framework generation algorithm generates the necessary artifacts, i.e., the test framework with its three main components (the proxy test model, the UTP artifact and the communication interface) for executing the model-based test cases in embedded systems. Hence, an expression for the time ($\Gamma_g^T$ ) and memory-size ($\Gamma_g^S$) complexity of the test framework generation algorithm can be defined as follows:

$$\Gamma_g^T = \beta + \gamma + \delta \tag{4.1}$$

$$\Gamma_g^S = \beta + \gamma + \delta \tag{4.2}$$

#### 4.5.1.a Test framework generation - Time complexity: $\Gamma_g^T$

An expression for the test framework generation time complexity in (4.1) can be derived as follows.

#### Proxy test model

The proxy test model component is created based on the classes participating in the association ends of the chosen SUT (lines:3-9 in algorithm 1). This has a processing complexity of $O(a \cdot c)$ (e.g. notation $a \cdot c$ denotes the product of $a$ and $c$).

However, among these classes, if the currently processed class is that of the chosen SUT, then a procedure to create the infrastructure to handle the test input data for the SUT in the proxy test model is invoked (lines:10-14 in algorithm 1). This procedure is discussed in algorithm 2 in section 4.4.3. The complexity of this processing (for creating the state chart of the SUT) depends mainly on

the number of events (e) associated with the chosen SUT ($\psi$). Therefore, the processing complexity for this algorithm is stated as $\psi \cdot O(e)$.

The proxy classes created in the proxy test model comprise of the functionality to interpret the incoming test results from the target debugger. On interpreting the test results, which comprise of data such as an event sent from a source to a destination class, the proxy test model generates this corresponding event between the respective proxy source and proxy destination classes. This functionality in the proxy test model is taken care of by the *generateEventConsumed()* operation (in the respective proxy class). The generation complexity involved during the creation of this function for every proxy class in the proxy test model can be stated as $O(e \cdot c)$.

Thus, the approximate time complexity for generating the proxy test model can be stated as $\beta = O(a \cdot c) + \psi \cdot O(e) + O(e \cdot c)$.

**UTP artifact**

The UTP artifact generation involves statements for creating the five elements of the UTP artifact component, namely, the instance of the SUT ($M_{sut}^{Test}$), the test component ($T_{comp}$), the test context element ($T_{context}$), the test configuration ($T_{config}$) and the test control ($T_{control}$). The series of steps involved in generating the UTP artifact component is discussed in algorithm 3 in section 4.4.4.

The creation of these UTP-based components involves applying a respective stereotype for the chosen element. Hence, the processing complexity for creating the UTP artifact component in the test framework can be stated as $O(n)$ (involving 'n' instruction sequences), i.e., $\delta = O(n)$.

**Communication interface**

The procedure for creating the communication interface (algorithm 4) generates the appropriate dependencies between the communication interface and the proxy classes in the proxy test model (lines:4-6 in algorithm 4). In addition, the communication interface component is also dependent on the test context element in the UTP artifact. Therefore, the processing complexity of these statements can be approximated to $O(n)$ (involving 'n' instruction sequences).

The algorithm 4 also adds the send ($T_{send}^{Com}$), receive ($T_{receive}^{Com}$) and decode ($T_{decode}^{Com}$) functionalities for the communication interface component of the test framework.

The processing complexity of the receive functionality, i.e., $T_{receive}^{Com}$ in *execute()* can be stated as $O(n)$ (involving 'n' instruction sequences). Similarly, the processing complexity of the send functionality in $T_{send}^{Com}$ (*sendToTrgtDbgr()*) can be approximated to $O(n)$ (involving 'n' instruction sequences).

However, the processing complexity of the decode functionality in $T_{decode}^{Com}$ (*decodeMsgFromTrgtDebg()*) is dependent on the number of events and classes associated with each class (i.e., each class in the association end of a chosen SUT in design model). As seen in the code-snippet provided in section 4.4.5.e, string compare statements are added to check the event name, the source of the event and the destination of the event. Hence, the processing complexity of $T_{decode}^{Com}$ can be stated as $O(e \cdot c \cdot c)$ (here, each class $c$ can be a source or a destination). Therefore the total processing complexity for creating the communication interface component is $\gamma = 3 \cdot O(n) + O(e \cdot c \cdot c)$.

Based on the values $\beta$, $\delta$ and $\gamma$, the approximate time complexity involved in the generation of the

test framework at the host computer can be stated as

$$\Gamma_g^T = O(a \cdot c) + \psi \cdot O(e) + O(e \cdot c) + O(n) + 3 \cdot O(n) + O(e \cdot c \cdot c) \tag{4.3}$$

Consider the following assumptions:

(a) The maximum number of classes in an association end is 2.

(b) For processing $n$ computational statements or instruction sequences (no nested-loops), the complexity is of the order $O(n) = n$.

Based on the above assumptions, the approximate *generation time complexity* ($\Gamma_g^T$) of the test framework algorithm is stated as $7n + n^2 + n^3$ $(2n + n + n^2 + n + 3n + n^3)$. In other words, $O(n) \approx n^3$. Note that in this complexity analysis, the number of SUTs supported per test framework is considered as one, i.e. $\psi = 1$.

***Inferences:***

From the theoretical estimate for the test framework generation-time complexity ($\Gamma_g^T$), the following inferences can be obtained. The time taken to generate the *UTP artifact* component is the least among the time taken for generation of the three components of the test framework. This is followed by the time taken to generate the *proxy test model* component. Whereas, the processing complexity of the $T_{decode}^{Com}$ operation in the *communication interface* component is expected to occupy a significant portion of the total test framework generation time as per the theoretical estimate. This can be attributed to the reasoning that the generation of the string compare statements for decoding the test results in the $T_{decode}^{Com}$ operation (i.e., *decodeMsgFromTrgtDebg()* function body) increases with an increase in the number of events and classes for a chosen SUT (section 4.4.5). These inferences based on the theoretical estimate of the test framework generation time complexity are recalled during the experimental evaluation of the parameters in the complexity measures in chapter 5.

#### 4.5.1.b Test framework generation - memory complexity: $\Gamma_g^S$

The approximate memory (size) complexity of the test framework generation algorithm is calculated by a reasoning similar to that for ascertaining time complexity. The *generation* memory complexity of this algorithm $\Gamma_g^S$ is stated as

$$\Gamma_g^S = \psi \cdot [O(a \cdot c) + O(e) + O(c \cdot e)] + O(n) + O(n) + O(e) \cdot O(a \cdot c) \tag{4.4}$$

The expression for $\Gamma_g^S$ (in 4.4) is arrived based on the following observations.

- The storage requirement during the generation of the proxy test model grows with the number of associations in a given SUT, and the classes in each association end ($\psi \cdot O(a \cdot c)$).
- During the proxy test model creation, when the processed class is that of the chosen SUT then a procedure to create the infrastructure/functionality to handle the test data in the proxy test model is invoked. This is created based on the functionality of the state chart of the SUT in the design model. The storage requirement for this is $\psi \cdot O(e)$.

- In addition, the test framework generation algorithm also invokes a procedure to create the functionality to decode the trace data in the test framework (section 4.4.2). The storage requirement for this is $\psi \cdot O(c \cdot e)$. Therefore the total storage requirement while generating the proxy test model is $\psi \cdot [O(a \cdot c) + O(e) + O(c \cdot e)]$.
- The storage requirement for the UTP artifact creation can be approximated to $O(n)$. This corresponds to the memory requirement for creating the elements in the UTP artifact component with a specific stereotype.
- The storage requirement for the TCP/IP communication interface component can be divided into two parts. First, the storage requirement to create the usage dependencies on the proxy classes. This can be approximated to $O(n)$ (involving 'n' instruction sequences). Second, the storage requirement for creating the decoding functionality in TCP/IP component grows with the number of classes associated with the chosen SUT $(a \cdot c)$ and the events in transitions $(e)$. Hence the storage requirement for the TCP/IP communication interface can be approximated to $O(n)$+$O(e) \cdot O(a \cdot c)$.

Consider the following assumptions:

(a) The maximum number of classes in an association end is 2.

(b) For given input data of size $n$ the approximate memory requirement (complexity) can be defined as $O(n) = n$.

Following the above reasoning and assumptions, the approximate test framework generation-memory complexity of the test framework algorithm $(\Gamma_g^S)$ is stated as $5n + n^2 + n^3$, i.e., $O(n) \approx n^3$. Note that in this complexity analysis the number of SUTs supported in the test framework is considered as one, i.e., $\psi = 1$.

***Inferences:***

Similar to the test framework generation-time complexity, the following insights can be obtained based on the theoretical estimate of the test framework generation-memory complexity. The memory requirement on the host computer for generating the UTP artifact component of the test framework is the least among the memory requirement for generating the three components of the test framework. This is followed by the memory requirement for generating the proxy test model component and then finally the communication interface component. Similar to the test framework generation-time complexity, the memory requirement for the decoding functionality in the communication interface component contributes to the significant increase in the storage requirement, as per the aforementioned theoretical estimates. These theoretical estimates are recalled and compared with the experimental results for the parameters in the complexity analysis in chapter 5.

In practice, the test framework generation algorithm is implemented and executed on a standard (desktop) computer for the automatic generation of the test framework for a chosen SUT. This gives flexibility in terms of choice of automation tools and the programming languages to be used. This is one among the important factors which may help mitigate the above test framework *generation* time and memory complexity measures.

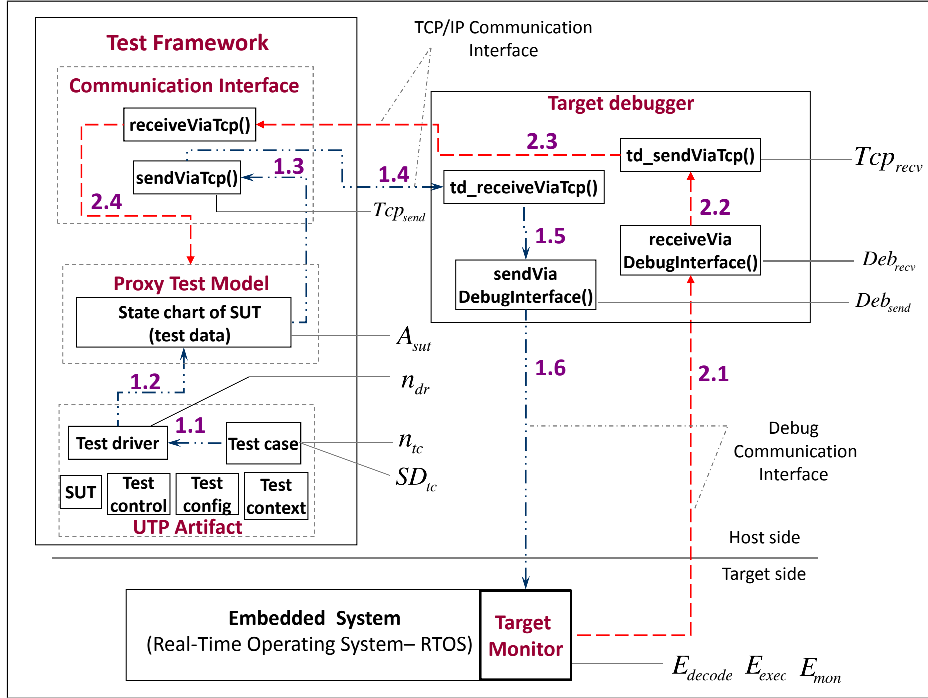



Figure 4.26: Execution parameters for executing model-based test cases in embedded systems

## 4.5.2 Runtime complexity

This section provides an analysis on the the *runtime* complexity measures for the proposed test framework approach while executing the test cases in the embedded system. This measure is calculated based on the processing complexity involved in executing the test cases in the embedded system and visualizing the results at the host side using the generated test framework (in the MBT tool). In this thesis, UML sequence diagram-based test case scenarios are considered during experimental evaluation.

The steps involved in executing the test cases in the embedded system using the proposed approach and the execution parameters for these steps are shown in Figure 4.26. In order to calculate this complexity measure, the following execution parameters are taken into consideration.

- $n_{tc}$ represents the number of sequence diagram test cases ($SD_{tc}$) to be executed in the embedded system.
- $n_{dr}$ denotes the total number of test driver triggers in a test case ($SD_{tc}$). $A_{sut}$ denotes the total number of actions taken in the state chart of the SUT in the proxy test model corresponding to triggers $n_{dr}$.
- $\Gamma_r^S$ and $\Gamma_r^T$ denote the approximate *runtime* memory and time complexity measures for executing the test cases on the embedded system using the discussed approach/test framework.

#### 4.5.2.a Time complexity: $\Gamma_r^T$

The *runtime* complexity can be calculated based on the processing complexity involved in executing the test cases in the embedded system using the proposed approach. The steps involved in executing

the test cases using the proposed approach is discussed in section 3.4 and the execution parameters defined for the various steps are shown in Figure 4.26. The parameters defined for the various steps are as follows:

- $Tcp_{send}$ denotes the total time taken to send $A_{sut}$ to the target debugger (steps 1.3 and 1.4 in Figure 4.26).
- $Deb_{send}$ is the time taken by the target debugger to convey the test data to the embedded system via the target monitor (steps 1.5 and 1.6 in Figure 4.26).
- $E_{decode}$ is the time taken to decode the incoming test data (from the target debugger) in the embedded system.
- $E_{exec}$ denotes the time to execute the test data in the embedded system.
- $E_{mon}$ is the time taken by the monitor to send the notifications (e.g. of test data execution in embedded system) via the debug interface to the target debugger (step 2.1).
- $Tcp_{recv}$ represents the time spent by the target debugger to send the test results to the test framework (step 2.3) via the TCP/IP communication interface.
- $Deb_{recv}$ denotes the time taken to send the test results (i.e., event execution results) to the target debugger via a given debug communication interface. The trace data is received by the target debugger and processed for interpreting the test results.

Based on these parameters, an expression for the *runtime* (time) complexity of the test framework generation algorithm can be defined. It is the sum of the processing and execution time taken to execute the given set of test cases for a chosen SUT using the test framework (in the host computer and in the embedded system). Hence it comprises of two parts and is defined as follows.

$$\Gamma_r^T = n_{tc} \cdot n_{dr} \cdot A_{sut} \cdot [\overbrace{Tcp_{send} + Deb_{send} + Deb_{recv} + Tcp_{recv}}^{\text{host computer}} + \overbrace{E_{decode} + E_{exec} + E_{mon}}^{\text{embedded system}}] \tag{4.5}$$

The expression for $\Gamma_r^T$ in (4.5) is derived based on the following reasoning:

**Host computer**

- The time consumed on the host computer for a given set of test cases $n_{tc}$, each with $n_{dr}$ test drivers involves invoking $A_{sut}$ actions (based on the input test data from the test cases) in the state chart of the SUT in the proxy test model (in the test framework package).
- For each of these actions, the test data is communicated by the TCP/IP communication interface in $Tcp_{send}$ time units and the target debugger in $Deb_{send}$ time units to the embedded system.
- Similarly, the test result (i.e., event consumed notification) is sent via the debug communication interface in $Deb_{recv}$ time units.

- The test data is interpreted by the target debugger and sent to the test framework in $Tcp_{rec}$ time units.
- Hence, the time consumed on the host computer is $n_{tc} \cdot n_{dr} \cdot A_{sut} \cdot [Tcp_{send} + Deb_{send} + Deb_{recv} + Tcp_{recv}]$.

**Embedded system**

- The time taken on the embedded system is a sum of the following. The time to decode the incoming test data from the target debugger $E_{decode}$, the time for executing the incoming $A_{sut}$ test data on the embedded system $E_{exec}$ and the time spent in the target monitor routine to send the notifications of the results (e.g. event consumed notification) to the target debugger by the target monitor $E_{mon}$.
- Hence, the total time taken on the embedded system is $n_{tc} \cdot n_{dr} \cdot A_{sut} \cdot [E_{decode} + E_{exec} + E_{mon}]$.

Consider the following assumptions.

(a) For processing $n$ computational statements or instruction sequence (no nested-loops) on the host, the complexity is of the order $n$ (i.e., $O(n) = n$).

(b) For executing $n$ instructions on an embedded system (running a real time operating system), the complexity is of the order $n'$ (i.e., $O(n) = n'$).

Based on the above assumptions, the approximate *runtime* (time) complexity ($\Gamma_r^T$) of the proposed approach for executing the test cases in the embedded system is

$$O(n) = n_{tc} \cdot n_{dr} \cdot A_{sut} \cdot [\overbrace{4n}^{\text{host computer}} + \overbrace{3n'}^{\text{embedded system}}]$$

i.e., $O(n) = n_{tc} \cdot n_{dr} \cdot A_{sut} \cdot [4n + 3n']$.

***Inferences:***

From the theoretical estimate for runtime-memory complexity (in (4.5)), the following inferences can be obtained. The time spent in the host computer (during the execution of the test cases) is a sum of the time to send, receive and decode the test data (test input, test results). When the host computer uses a non-real time (and non-deterministic) operating system such as Windows and Linux, the time taken to send, receive and decode the test data can be expected to vary over several iterations (i.e., even under the same setup).

On the other hand, the time spent in the embedded system for executing the test cases is only that of decoding the test input data, executing it and sending notification to the host about the target behavior. Since the embedded system uses a deterministic RTOS (e.g. OORTX-RXF used in the prototype), the time spent for the aforementioned actions remains constant over any number of iterations (i.e., under the same setup). Therefore in the proposed approach, the time spent to execute the test cases in the target is only that of decoding the test input data, executing it and sending notification to the host about the target behavior by the target monitor routine.

#### 4.5.2.b Memory complexity: $\Gamma_r^S$

The approximate (*runtime*) memory complexity $\Gamma_r^S$ of the test framework generation algorithm is calculated by a reasoning similar to that used for determining $\Gamma_r^T$. Based on $n_{tc}$ for a chosen SUT, the following parameters are defined for determining $\Gamma_r^S$:

- $S_{TF}$ is the storage requirement for the automatically generated *test framework* on the host computer. The various components of the test framework aid the test case execution process (i.e., for executing model-based test cases) at the host computer.

- $S_{TD}$ is the storage requirement for the *target debugger* on the host computer. The target debugger helps in communicating the test data (e.g. test input and test results) between the test framework and the target monitor.

- $S_{TM}$ defines the total memory requirement on the embedded system for the *target monitor*. This includes the functionality to receive the test input data from the target debugger and to send the test results (e.g. event consumption notification) to the target debugger on the host computer.

Based on these parameters, the approximate *runtime* memory complexity (for executing the test cases on the embedded system using the proposed approach/test framework) is stated as

$$\Gamma_r^S = \overbrace{\overbrace{[A_{sut} \cdot S_{TF}] + [n_{tc} \cdot n_{dr}]}^{\text{test framework}} + \overbrace{S_{TD}}^{\text{target debugger}}}^{\text{host computer}} + \overbrace{S_{TM}}^{\text{embedded system}} \tag{4.6}$$

The expression for $\Gamma_r^S$ in (4.6) is derived based on the following observations:

**Host computer**

- The storage requirement during the execution of the test cases for a given SUT using the proposed approach can be divided into two parts, namely, the storage requirement in the host computer and the storage requirement in the embedded system.

- The storage requirement on the host computer can be divided into the storage requirement for the automatically generated test framework and the storage requirement for the target debugger.

- Given a chosen SUT, the test framework generation algorithm generates a corresponding test framework for executing the model-based test cases. This test framework comprises of the test input data and also the necessary test artifacts (i.e., infrastructure support) for executing the model-based test cases in embedded systems. The memory requirement of the test framework alone without/before the specification of the test cases is given by $A_{sut} \cdot S_{TF}$.

- Once the test framework is generated and built successfully, a varying number of test cases can be specified in the test framework. Upon specification, the test harness is updated and the additional code is generated for executing the $n_{tc}$ test cases which contain $n_{dr}$ test drivers, by the MBT tool. This memory requirement corresponds to $n_{tc} \cdot n_{dr}$.

- Thus the memory requirement for the test framework on the host computer comprises of a fixed component $A_{sut} \cdot S_{TF}$ and a varying component $n_{tc} \cdot n_{dr}$ based on the number of test cases specified in the test framework. Hence, the total memory requirement of the test framework on the host computer is stated as $A_{sut} \cdot S_{TF} + n_{tc} \cdot n_{dr}$

- The storage requirement for the target debugger is however a fixed memory requirement of $S_{TD}$. This corresponds to the memory requirement of the target debugger executable or its corresponding DLL file created at the host computer.

- Hence the storage requirement in the host computer is stated as $[A_{sut} \cdot S_{TF}] + [n_{tc} \cdot n_{dr}] + S_{TD}$.

**Embedded system**

- The software-based runtime monitoring approach introduced in this thesis, aids in inserting the test stimuli to the target. It also sends the test case execution result (e.g. event consumed notification) to the host computer. Hence the storage requirement for executing the test cases on the embedded system is $S_{TM}$.

- The target monitor is a static/constant software routine with a bounded and measurable overhead. The target monitor and its memory requirement is independent of the given embedded software application size and complexity. Hence, the memory-size complexity for executing the model-based test cases on the embedded system has a fixed memory requirement of $S_{TM}$ on the embedded system.

Consider the following assumptions:

(a) For a given input data of size $n$, the approximate memory requirement on the host computer can be defined as $O(n) = n$.

(b) The fixed component in the memory requirement for the test framework $(A_{sut} \cdot S_{TF})$ can be approximated to a constant $C_{tf}$.

(c) The memory requirement of the target debugger also does not vary during the runtime, i.e., during the test case execution process. Hence, the memory requirement of the target debugger during runtime can be approximated to a constant $C_{td}$.

(d) The fixed memory requirement of the target monitor $(S_{TM})$ on the embedded system can be approximated to a constant $C_{tm}$.

Then the approximate memory complexity during runtime $(\Gamma_r^S)$ for executing the model-based test cases in the embedded system is

$$O(n) = \overbrace{C_{tf} + O(n^2) + C_{td}}^{\text{host computer}} + \overbrace{C_{tm}}^{\text{embedded system}}$$

***Inferences:***

Among the parameters in the runtime memory complexity, the memory requirement in the embedded system for executing the test cases can be considered as the most significant one. Based on the above

described complexity analysis, it is clear that the memory requirement in the embedded system for executing the test cases is only that of the target monitor. Since the target monitor does not vary based on the number of test cases to be executed or their complexity, there is only a fixed memory requirement on the target for executing the test cases.

These theoretical estimates from the complexity analysis are recalled during an experimental evaluation/analysis of the parameters in the complexity measures in the next chapter.

# Chapter 5

# Experimental Evaluation

In this chapter, an experimental evaluation of the proposed integrated model-based approach and test framework is discussed. This pertains to challenges (1), (2), (3) and (4), wherein an experimental evaluation of an integrated model-based approach and test automation towards executing the model-based test cases in the embedded system is envisaged.

Towards this direction, a prototype implementation of the *target monitor* and the *target debugger* along with their performance metrics are expected to provide insights on the overhead involved in executing the model-based test cases using the proposed approach. Furthermore, significant inferences are expected to be obtained for the parameters in the complexity analysis (introduced in section 4.5), based on an experimental analysis of these parameters (in this chapter).

An overview of the tools, languages and example application scenarios used in the prototype and experimental evaluation of the proposed approach is provided in Figure 5.1.

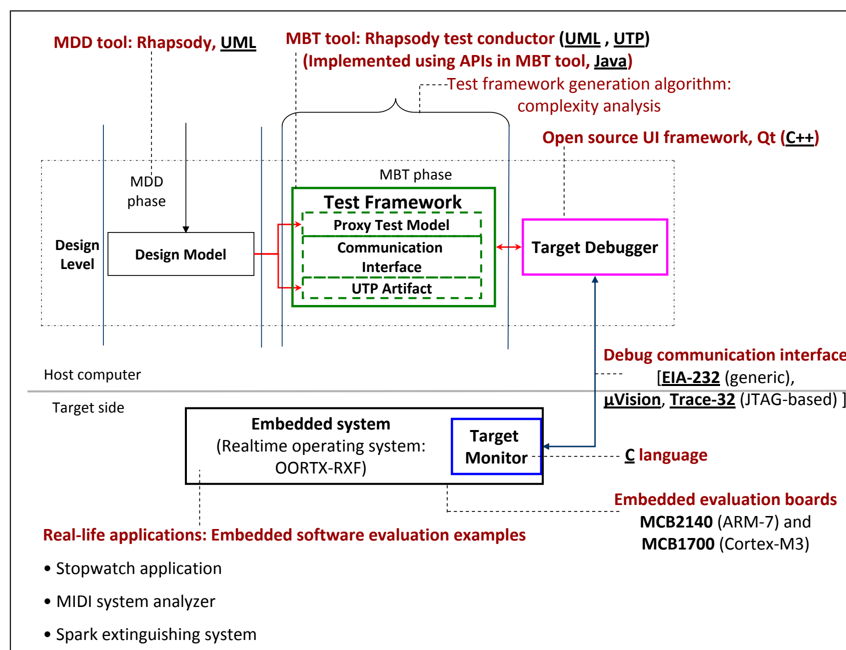



Figure 5.1: Tools, languages and example application scenarios used during prototype and experimental evaluation

This chapter is organized as follows. In section 5.1, a prototype implementation of the proposed target monitor (i.e., the runtime monitoring approach-challenge (3)), its performance metrics and the applicability of the proposed target debugger are elaborated.

In section 5.2, the specification of the model-based test cases in the automatically generated test framework for the MIDI system analyzer case study is outlined. The series of steps involved in executing the model-based test cases (e.g. sequence diagram-based test cases) using the automatically generated test framework for the MIDI system analyzer case study are discussed (challenge (4)). An analysis of the results based on the experimental evaluation of the proposed approach is also provided. Note that the MIDI system analyzer case study, its system design model and the automatically generated test framework have been introduced in chapter 4.

An experimental analysis of the parameters in the complexity analysis (introduced in section 4.5) is provided in section 5.3. In addition, a discussion based on the experimental results on example application scenarios is provided in section 5.3 (challenges (3) and (4)). Thus, the time and memory-size complexity involved during the generation of the test framework and during the execution of model-based test cases (runtime) are discussed in detail in section 5.3.

## 5.1 Target debugger and target monitor

In order to enable test automation and model-based test case execution without introducing significant overhead in the embedded system (challenge (3)), the test framework makes use of two components, namely a *target debugger* on the host computer and a *target monitor* at the embedded system. A brief outline of the functionality of the target debugger and the target monitor in the proposed approach is provided below. Their respective prototype implementation, during the experimental evaluation, is discussed in section 5.1.1 and section 5.1.2.

#### Target debugger

To support a runtime monitoring mechanism which is envisaged to be a generic routine on the embedded system, prior knowledge about the embedded software executed on the target should be explicitly available at the host computer. For example, when the test case execution process is carried out at the host computer and only the test input data is sent to the target (as in the case of the proposed approach), prior information regarding the components of the test input data must be made available beforehand at the host computer. Similarly, to decode the trace data from the target, prior information regarding the components of the test result must be available at the host computer. Such a decoding (and mapping) program at the host computer is denominated as the target debugger, in this thesis. With the aid of a information marshalling and decoding agent at the host computer, namely the target debugger, additional monitoring overhead at the embedded system can be eliminated. Now the overhead involved in the embedded system will be (only) that of the software-based runtime monitoring routine (denominated as the target monitor) to insert the test input data/stimuli and convey the test results (i.e., the trace data) to the host computer via a debug communication interface (Figure 5.2). A prototype implementation of the target debugger developed in this thesis is discussed in section 5.1.1.

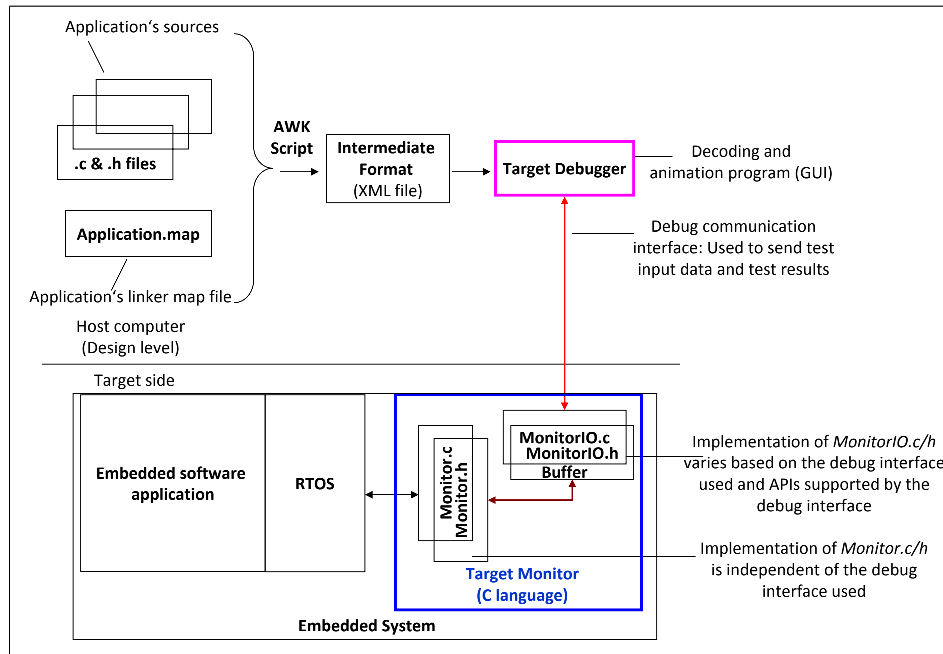



Figure 5.2: XML file creation at the host and target monitor design at the embedded system

### Target monitor

The runtime monitoring methodology considered in this thesis is envisaged to be time and memory-size aware, i.e., measurable and bounded overhead for the target monitor. In addition, the target monitor is envisaged to be a generic and a minimal (software) routine in the embedded system. At the same time, the target monitor is expected to support the model-based test case execution and be able to sufficiently observe the target behavior (challenge (3)). In line with these goals, the requirements and a proposal of a generic, software-based runtime monitoring methodology is provided in section 3.3.

In the proposed approach, the target monitor in the embedded system has two main functionalities. First, the test input data is injected to the embedded system with the aid of the target monitor in a pre-defined frame format (Figure 3.5). Second, the test results (e.g. event consumed notification) is sent to the host computer (i.e., the target debugger) in a pre-defined frame format (Figure 3.6). Based on this, the functionality of the target monitor, can be grouped into two main aspects (Figure 5.2), namely:

- Function 1: Communicating with the RTOS framework
- Function 2: Communicating with the host computer

Therefore in the prototype implementation of the target monitor, the functionality of the monitoring routine can be modularized in software routines such as *Monitor* (for function 1) and *MonitorIO* (for function 2) as seen in Figure 5.2. A prototype implementation of the target monitor, based on the aforementioned functionalities, is discussed in section 5.1.2.

### 5.1.1 Prototype implementation of the target debugger

There exists several choices of programming languages (e.g. Java, C++) and tools (e.g. Qt [87], wxWidgets [120], GTK+ [106]) for the implementation of a decoding and animation program (GUI) such as the target debugger. The cross-platform user interface framework Qt [87] (with open source libraries in C++) can be considered as one among the best industry-standard alternatives for the implementation of a decoding and animation program, such as the target debugger on the host computer. Hence, in the prototype, the target debugger is implemented in the programming language C++ using the user interface framework Qt. The target debugger comprises of a decoding program and an animation program, in the prototype implementation, which are described in sections 5.1.1a and 5.1.1.b respectively.

#### 5.1.1.a Decoding program in the target debugger

The decoding program in the target debugger has two main functionalities, namely, (a) it is used to map the test input data (from the test framework) to object addresses (used by the target) and (b) it is used to decode the trace data from the target (i.e., by mapping the high-level/design-level names such as class names, event names and state names to the object addresses).

In the prototype implementation of the target debugger (using Qt), the decoding and mapping functions are performed by the target debugger by making use of a XML file at the host computer. This XML file, in turn, is generated using the following procedure (Figure 5.2). During the compilation of the (embedded software application) system code (obtained from the MDD phase), an AWK script parses the linker map file, source files, header files (of the application) and creates a mapping between the design-level names (such as class, event and state names) at the host and the object addresses used at the target. This is stored in an intermediary format such as the XML file as seen in Figure 5.2. Thus, the context information about the generation of the binary executable is available in the XML file created at the host computer.

The decoding program in the target debugger makes use of this XML file (Figure 5.2) to decode and interpret the test input (from the test framework) or the test results (e.g. trace data denoting event consumed notification). The advantage/significance of the decoding functionality in the target debugger is discussed in section 3.3.2 (Figure 3.7 and Figure 3.8).

To gain further understanding, consider the XML file generated at the host computer for the MIDI system analyzer embedded software application. An excerpt of this XML file (with the relevant data for explaining the examples below) for the MIDI system analyzer case study is shown in Figure 5.3. As seen in Figure 5.3, the XML file comprises of information about both the embedded system (hardware) used and the application software running on the embedded system. This XML file is generated (at the host) by parsing the linker map file, source files and header files from the source code automatically generated by the MDD tool (Rhapsody) for the MIDI system analyzer embedded software application. The following describes the data stored in the XML file (Figure 5.3).

1. **Endianness and types:** Information about the *endianness* and *types* used in the embedded system is stored in the XML file as shown in Figure 5.3. This is obtained by parsing the source files and the header files for a given application scenario. This information may be required at the host to understand the nature of the incoming trace data from the target.

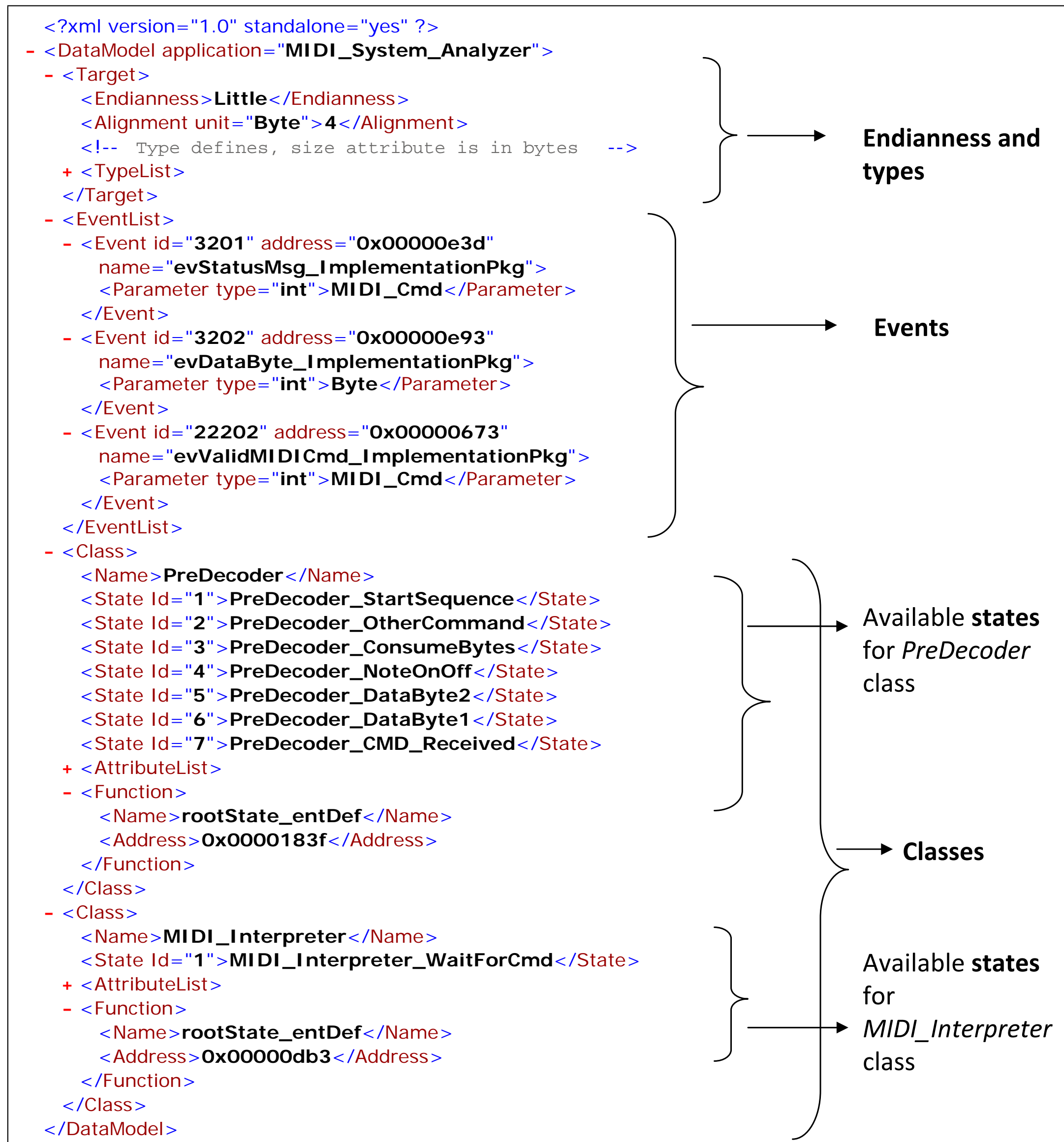



Figure 5.3: An excerpt of the XML file generated (using the procedure shown in Figure 5.2) at the host computer for the MIDI system analyzer application example

Following this, the information required to decode the components of the trace data from the target is generated in the XML file. The three main elements obtained by parsing the linker map file, header file and source files for a given application are the list of *events* and *classes* for a given application and the list of *states* and *attributes* for each class.

2. **Events:** The list of events for a given application example are obtained by parsing the application's header files. The following code shows a (subset of the) list of events defined for the MIDI system analyzer example.

```
1 #define evStatusMsg_ImplementationPkg_id       3201
2 #define evDataByte_ImplementationPkg_id        3202
3 #define evValidMIDICmd_ImplementationPkg_id    22202
```

In the design model, the MIDI system analyzer is modeled and encapsulated in a package named *ImplementationPkg*. Therefore, the above data is available in the *ImplementationPkg.h* header file. As seen in the code snippet, each event is associated with an *event id*, in the automatically generated code (in the MDD tool Rhapsody). Note that the event name is followed by the package name in which the event is defined (e.g. *evStatusMsg_ImplementationPkg_id* ).

The mapping between the event id and its corresponding address is created by parsing the linker map file. For the events listed above, the mapping to object addresses is obtained from the following information in the linker map file.

```
1 RiC_Create_evStatusMsg       0x00000e3d   Thumb Code    ImplementationPkg.o
2 RiC_Create_evDataByte        0x00000e93   Thumb Code    ImplementationPkg.o
3 RiC_Create_evValidMIDICmd    0x00000673   Thumb Code    ImplementationPkg.o
```

Thus, given the event id and the object addresses, a mapping between the event name, event id and its address can be created (and stored in XML format) as follows.

```
1 <Event id="3201" address="0x00000e3d" name="evStatusMsg_ImplementationPkg">
2 <Parameter type="int">MIDI_Cmd</Parameter>
3 </Event>
```

Thus, this mapping is provided for all the events in a given embedded software application. This mapping between the event name, id and the object addresses created for the three events *evStatusMsg(), evDataByte()* and *evValidMIDICmd()* are shown in the automatically generated XML file, in Figure 5.3.

3. **Classes:** When each class is defined with a default state chart, a function *rootState_entDef()* is generated for the state chart of the corresponding class, by the MDD tool Rhapsody. Thus, a class is identified by the address of the corresponding *rootState_entDef* function which in turn is generated when a class is specified with a state diagram (which can also be empty).

```
1 rootState_entDef    0x0000183f   Thumb Code    PreDecoder.o
2 rootState_entDef    0x00000db3   Thumb Code    MIDI_Interpreter.o
```

This aspect is tool specific, i.e., pertains to the code generation technique used in the MDD tool used in the prototype implementation namely, Rhapsody. Therefore, if any other MDD tool is employed, the object address of each class can be obtained using other mechanisms or parsing the respective code generated for state charts. For example, a function *stateTriggerHandler* in the code generated by another MDD tool [23] provides the same information as that of the *rootState_entDef* function in the code generated by the MDD tool Rhapsody.

Similarly, the list of attributes and instances for each class can also be obtained by parsing the source and header files for a given application. The XML file shown in Figure 5.3 illustrates the pertinent data about the two main classes, namely the *PreDecoder* class and the *MIDI_Interpreter* class for the MIDI system analyzer case study.

4. **States:** The various states of a state chart in a class are associated with an identifier in the generated code. For example, the possible states of the *PreDecoder* class are obtained from the generated header file (*PreDecoder.h*), which comprises of the following implementation (for defining the states).

```
1 enum PreDecoder_Enum {
2       PreDecoder_StartSequence=1,
3       PreDecoder_OtherCommand=2,
4       PreDecoder_ConsumeBytes=3,
5       PreDecoder_NoteOnOff=4,
6       PreDecoder_DataByte2=5,
7       PreDecoder_DataByte1=6,
8       PreDecoder_CMD_Received=7
9       };
```

Based on the above data, the list of states for the *PreDecoder* class is stored in the XML file as shown in Figure 5.3.

Significant overhead involved in sending the trace data back and forth between the host computer and the target is avoided by the use of the XML file at the host computer and the pre-defined frame format for notifications (section 5.1.2.b). Thus, the decoding program at the host computer makes use of this XML file to decode and interpret the incoming trace data which is sent via a debug communication interface as seen in Figure 5.2. It also uses the XML file to map the test input data comprising of high-level names (e.g. class names) to object addresses. Then the test input data, available as a mapping between design-level names and object addresses, is injected in the embedded system by the target monitor. Note that an illustration of this technique and the applicability of the target debugger in the context of the proposed test framework approach is discussed in section 3.3. To gain further understanding, the mapping of test input data and test results by the target debugger using the XML file (Figure 5.3) for the MIDI system analyzer example is elaborated in the following.

**Mapping of test input data and test results to the data in the XML file**

First, consider the test input data injected by the test framework to the target via the target debugger. As discussed in chapter 4, this test input data is available from the state chart of the SUT in the test framework. Consider an example to inject an event *evStatusMsg()* with parameter *144* from the test environment *TestEnv* to an instance of the *PreDecoder* class, sent by the test framework to the target debugger. The test input data (from the test framework to the target debugger) is sent in a pre-defined format (Figure 3.5) and comprises of the following data string for the above example, *<evStatusMsg TestEnv itsPreDecoder1 144>*. This test input data is sent by the test framework (using the communication interface component) to the target debugger. The target debugger maps this test input data to object addresses based on the data in the XML file in Figure 5.3. Thus the test input data from the target debugger to the embedded system corresponding to the above example is *<0x00000e3d 0x01 0x0000183f 0x90>*. Here, a hexadecimal value of *0x01* corresponds to the string token *TestEnv* representing the test environment.

Similarly, the trace data sent by the embedded system (via the target monitor) to the target debugger for the above example (i.e., corresponding *event consumed notification* from the target) is *<0x00000e3d 0x01 0x0000183f 0x00405010 0x07 0x90>*. These values are mapped to their respective design-level names by the target debugger using the XML file. For the above example, the mapped/decoded test result from the target debugger is *<evStatusMsg TestEnv itsPreDecoder1 230 CMD_Received 144>*. The current time value (i.e., 230) represents the difference in time between the current notification (for event consumed) obtained at the target debugger and the previous immediate

notification. Thus, the trace data comprising of object addresses (hexadecimal values) are mapped to their corresponding high-level names using the XML file at the host. The trace data decoded by the target debugger is sent to the test framework, for visualizing the test results in the MBT tool as UML diagrams.

Therefore, by making use of the XML file and the pre-defined frame format for notifications (Figure 5.7), significant overhead involved in sending the test input/test results between the host and the target is reduced. Since the decoding and mapping functionality performed at the target debugger (i.e., moved to the host computer), the overhead involved in performing the decoding/mapping operations at the embedded system is eliminated (challenge (3)).

#### 5.1.1.b Target debugger-Graphical User Interface (GUI)

The animation program in the target debugger, on the other hand, is used to draw UML diagrams (automatically) based on the decoded trace data in real time. Therefore, based on the decoded trace data at the target debugger, a model-based visualization of the target behavior can be realised at the host computer (design level), using the target debugger-GUI. This is an additional novel application of the proposed target debugger, foreseen in this thesis.

The example application scenarios considered in this thesis are modeled using UML during the design phase. Hence, the decoded trace data is visualized as UML diagrams by the animation program in the target debugger GUI at the host computer. Therefore, in the prototype implementation the target behavior is visualized in real time using models comprising of UML diagrams (e.g. UML sequence and timing diagrams) at the design level.

The target debugger GUI developed as a part of the prototype implementation in this thesis, is shown in Figure 5.4. The target debugger GUI comprises of three blocks/areas (a), (b) and (c). In Figure 5.4-(a) the classes, objects, states and attributes available in the embedded software running on the target are displayed. Figure 5.4-(b) displays the sequence of events and the temporal behavior of the target using UML sequence diagrams with time stamps (sequence diagram tab) and UML timing diagrams (timing diagram tab). The timing diagram notation used is the value lifeline notation or the concise notation. Figure 5.4-(c) shows the reconstructed messages on the host side. The target behavior is reconstructed at the host computer with the aid of the back annotated data received from the embedded system and the XML file generated at the host computer. Further details about the target debugger GUI and its application are also available in already published works such as [49], [50]. An example of visualizing the MIDI system analyzer behavior using UML diagrams in the target debugger GUI is outlined below.

Visualizing the MIDI system analyzer behavior in real time using UML sequence diagram (with time stamps) in the target debugger GUI is illustrated in Figure 5.5. The sequence diagram shown in Figure 5.5 is visualized in the field (b) of Figure 5.4.

For the chosen MIDI system analyzer example, for one key press (and subsequent release) on the MIDI keyboard five messages are expected to be generated. In other words the events are generated in the sequence *evStatusMsg(), evDataByte(), evDataByte()* for a key press and *evDataByte(), evDataByte()* are generated for the corresponding key released on the MIDI keyboard. These messages are generated by an instance of the *PreDecoder* class (e.g. *itsPreDecoder1*). Similarly, on receiving a sequence of key press and key release messages, an instance of the *MIDI_Interpreter* class generates a valid MIDI message, represented by the event *evValiDMIDICmd()*. These events occurring on the

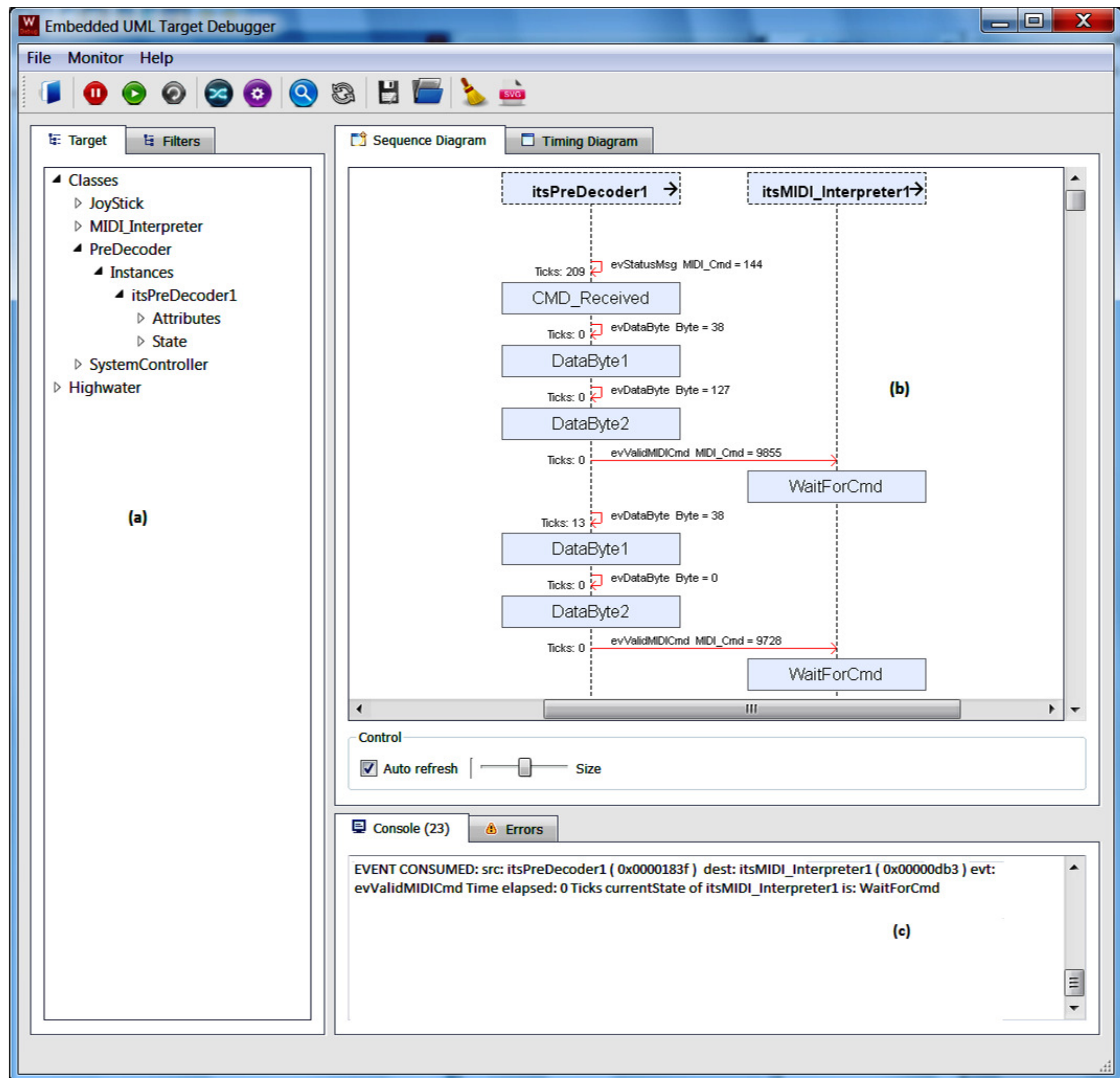



Figure 5.4: Screenshot of the target debugger GUI. Sequence and timing diagrams shown in Figure 5.5 and Figure 5.6 respectively are visualized in the area (b) in Figure 5.4.

target are visualized in real time using UML sequence diagram at the target debugger GUI. This is illustrated in Figure 5.5. Note that, Figure 5.5 represents the MIDI system behavior visualized as UML sequence diagram in the target debugger GUI, i.e., in Figure 5.4-(b). In the target debugger GUI, event receptions, state changes of instances of objects on the target and the time elapsed between the various events occurring on the target are also displayed, as seen in Figure 5.5. As evident from Figure 5.5, the time elapsed between the individual events on the target is also displayed in the UML sequence diagram. This feature is a novel enhancement to existing UML sequence diagrams. This is achieved using the proposed approach by decoding and interpreting the incoming trace data to the host (which also consist of temporal information from the target) in this thesis. The time elapsed between the events, displayed in the UML sequence diagram, is used for the manual verification of temporal constraints (i.e., non-functional requirements), which is discussed in section 5.3.2.b.

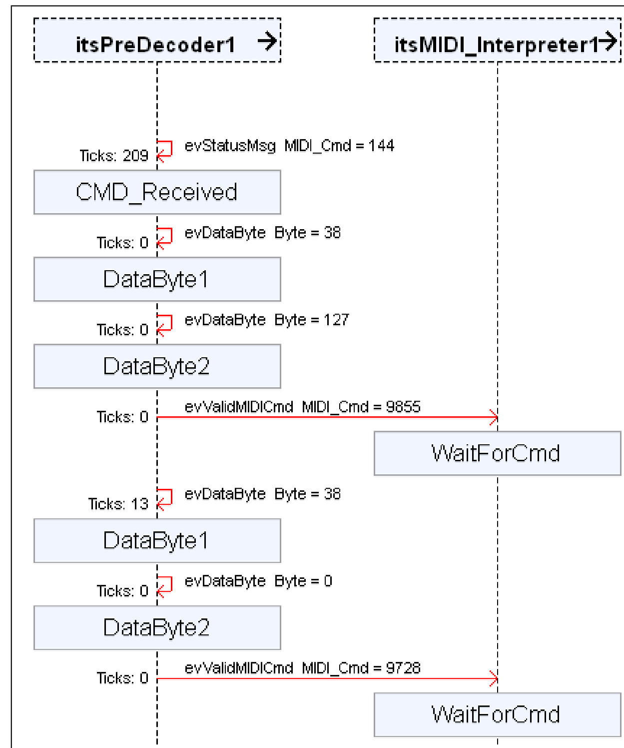



Figure 5.5: Visualizing MIDI system behavior in real time using UML sequence diagram in the target debugger GUI (in the area (b) in Figure 5.4)

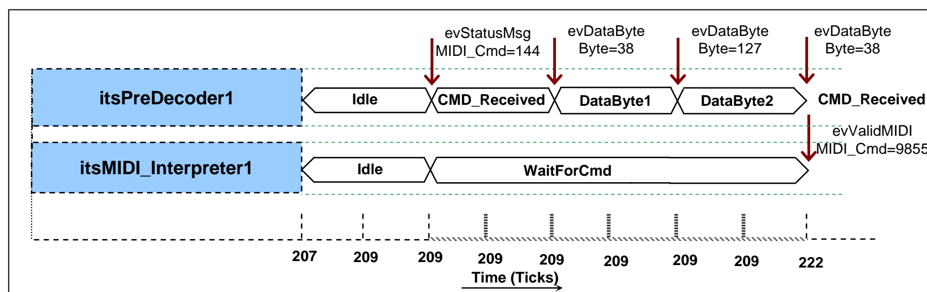



Figure 5.6: Visualizing MIDI system behavior in real time using UML timing diagram in the target debugger GUI (in the area (b) in Figure 5.4)

Similar to visualizing the target behavior in real time using the UML sequence diagrams, UML timing diagrams can also be employed to visualize the target behavior in real time. Figure 5.6 illustrates the target behavior on receiving a sequence of key press messages in the MIDI system analyzer case study, using UML timing diagram. The timing diagrams are displayed in the target debugger GUI in the area (b), i.e., in Figure 5.4-(b), in the timing diagram tab. The timing diagram notation used in the target debugger GUI is the value lifeline notation or the concise notation. Note that a brief introduction to UML sequence and timing diagrams is provided in chapter 2 (section 2.1.1.b).

### 5.1.2 Prototype implementation of the target monitor

In this thesis, the system code (executed on the embedded system) generated during the MDD phase is in the programming language C (e.g. as in the MIDI system analyzer). Hence the software-based runtime monitoring routine is also envisaged to be implemented in the programming language C. Then, the target monitor implementation (in the prototype) is modularized based on the aforementioned two functionalities (Figure 5.2) in software routines such as *Monitor.h/.c* and *MonitorIO.h/.c* respectively.

In the prototype, the *Monitor.c* routine is envisaged to be invoked in the RTOS framework for consumed events. The functionality for communicating with the RTOS, available in *Monitor.c*, is independent of the debug communication interface under consideration (Figure 5.2). The *Monitor* routine in turn uses the functions in the *MonitorIO* routine to send and receive data between the host computer and the embedded system via a debug communication interface. Whereas the *Monitor* routine communicates with the RTOS framework (e.g. by invoking functions), the *MonitorIO* is used for communicating with the host computer using a debug communication interface. The *MonitorIO* routine can then be implemented as a configurable routine based on the APIs and functionalities available in a given debug communication interface (e.g. EIA-232 [5] or JTAG-based [52]). The monitor implementation may also use a configurable buffer to handle the trace data.

#### 5.1.2.a Choice of the RTOS for the prototype/experimental evaluation

Since the experimental evaluation (in this thesis) is carried out in resource constrained embedded system examples (challenges (1), (2), (3)), an appropriate RTOS suitable for resource constrained embedded systems has to be chosen. An operating system framework optimized for usage in small embedded systems (i.e., with limited memory size), namely OORTX-RXF (Object Oriented Real-Time Execution Framework) [118] is used in the prototype evaluation. Whereas, other RTOS frameworks suitable for resource constrained embedded systems such as embOS [92] and CMX-RTX [14] can be used as the underlying RTOS for a given application scenario. In other words, the proposed approach in this thesis is not only independent of the modeling language (e.g. UML) and the language in which the system code is generated (e.g. C), but also independent of the RTOS used for a given application scenario.

The OORTX-RXF is an event-driven, fully configurable runtime environment and RTOS especially optimized for its usage in embedded systems with limited resources. In the prototype implementation, the target monitor routine is bundled with the underlying RTOS framework, i.e., OORTX-RXF. The target monitor is used to send notifications about the target behavior to the host computer. The OORTX-RXF being an event-driven system, the various operations are realized at the target by generating and consuming events at the OORTX-RXF. For example, the target monitor is invoked after an event is generated and consumed at the target (in the OORTX-RXF), to send an event consumed notification to the host computer (via a debug interface).

#### 5.1.2.b Functions supported by the target monitor

The *Monitor.h/.c* routine in the target monitor is envisaged to be independent of the debug communication interface used (Figure 5.2). It comprises of operations to handle the monitoring and debugging functionalities (bi-directional) between the host and the embedded system. For instance, such operations include injecting event (e.g. as test stimuli) from the host to the target and sending event

consumed notifications (e.g. as test result) to the host computer. The list of operations supported by the *Monitor* routine (in the prototype) and their description is provided in Table 5.1. For example, after an event is processed and dispatched to its respective receiver in the embedded system, *Monitor_sendEvent(unsigned int* pEventData)* function in the *Monitor.c* is used to notify the host about the event consumption at the embedded system. This function in the *Monitor* routine in turn invokes the respective function in the *MonitorIO* routine to send the trace data to the host computer.

Table 5.1: Operations supported by the *Monitor* routine in the target monitor

| **Operation** | **Description** |
|---|---|
| *Monitor_sendInit(void *me, void * rootState_entDef)* | Used to send trace data about new object creation in the embedded system. The "*me*" pointer corresponds to the address of the object and "*rootState_entDef*" corresponds to address mapping of the object to class. |
| *Monitor_sendDestroy(void *me)* | Used to send trace data about object destruction in the embedded system. |
| *Monitor_sendError(void)* | Invoked for error notifications. |
| *Monitor_performCmd(void)*§ | Invoked by the host computer, when the host is injecting an event to stimulate the system. It can also be used to retrieve specific trace data from the embedded system. |
| *Monitor_sendEvent(unsigned int* pEventData)* | Whenever an event is consumed at the embedded system, this function is used to send *event consumed notification* to the host computer. The parameter "*pEventData*" is a pointer to an integer array. It comprises of the data about the consumed event. |
| *Monitor_sendBufferReset(void)* | Sends notification to the host computer that the target monitor buffer is reset. |

§: This is the only function invoked by the host computer. The remaining functions are used by the embedded system for sending the respective notifications.

The two main functions supported by the *MonitorIO* routine are *int Monitor_IO_PutByte (int ch)* and *int Monitor_IO_GetByte (int ch)*. The former function is used to send one byte over the debug interface to the host computer. The latter is used to receive one byte over the debug interface from the host computer. These functions are implemented based on the APIs available for each debug interface.

The target monitor is thus modularized to minimize the variations that may be introduced by the different debug interfaces used (Figure 5.2). Thus, the functionality for communicating with the RTOS framework (Function 1), implemented in the *Monitor* routine is independent of the debug communication interface used. On the other hand, the functionality for communicating with the host computer (Function 2) implemented in the *MonitorIO* routine varies for different debug interfaces (Figure 5.2). This is mainly based on the APIs available for each debug interface and the functionality that it provides. It is also dependent on the features supported in the respective microcontroller family for the given target system. For the proposed approach, a generic EIA-232 (serial/UART) interface [5] and two industry standard JTAG-based interfaces are evaluated.

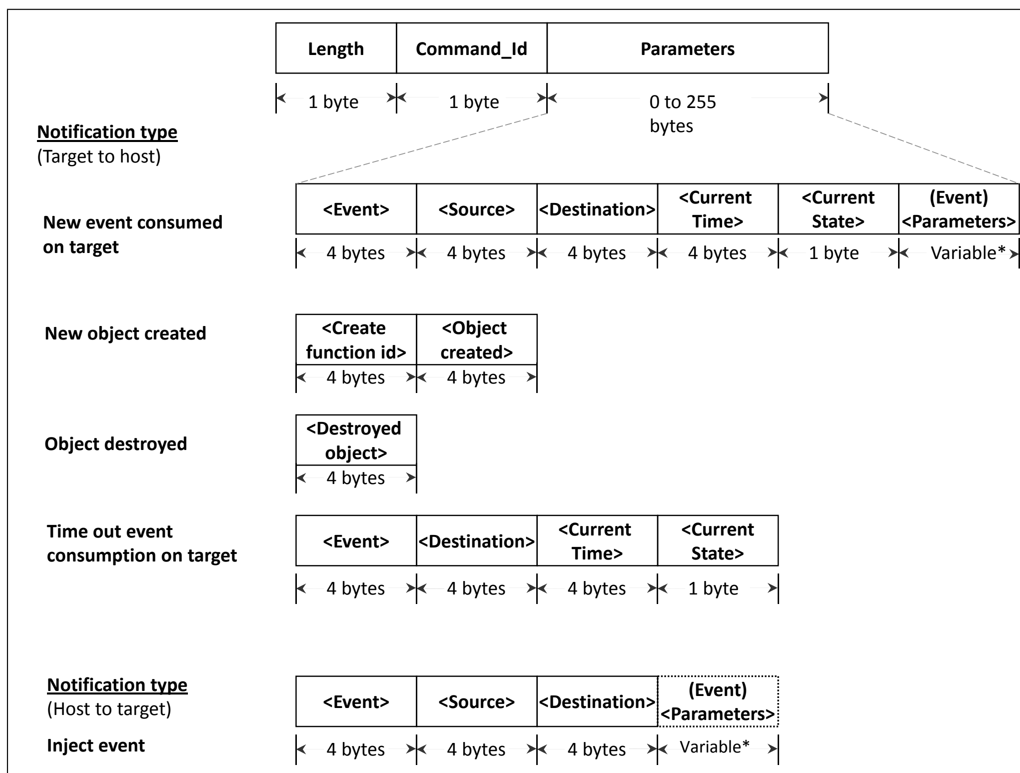
### 5.1.2.c Target monitor notifications

The pre-defined frame format for the test result notification from the target monitor (i.e., an *event consumed notification*) to the host computer is introduced in section 3.3. In the prototype, the target monitor is also used to send notifications such as object creation/destruction and time out event consumption at the target and status about reaching an error handler in the target.

Therefore, there arises a need for a mechanism to identify and distinguish between the various notifications from the target monitor to the target debugger. Hence, a mechanism to send the trace data from the target in a communication protocol-like format is perceived in the prototype implementation. Such a format usually comprises of a header and a data field (payload). The header field consists of information about the type of data sent (e.g. in this case, the notification type) and the length of the data (e.g. the length of the notification). Then the payload/data field comprises of the actual notification. The development of such a format is based on the following design considerations (in the prototype). They are,

- Compactness of the protocol-like format
- Minimum number of operations on the target and
- Extensibility of the protocol-like format.

The resulting frame format is shown in Figure 5.7. The three fields in the frame format (in the




*This is variable based on the type of the parameter for an event, e.g. 1 byte for char, 4 bytes for integer parameter, etc.

Figure 5.7: Target monitor frame format and parameters for various notifications

prototype implementation) are the "length", "command_id" and "parameters" (Figure 5.7). In the prototype, the "length" field is mandatory, one byte in length and indicates the length of the parameters. The mandatory "command_id" field is also one byte in length. It denotes the command corresponding to the frame sent. The "parameters" field is optional and can be between 0 and 255 bytes in length. It denotes the data about the current command. The minimum length of the monitor frame is two bytes (1 byte each for *length* and *command_id*).

Using the aforementioned frame format for notifications, the target monitor sends pertinent data about the embedded system behavior, in real time, to the host computer. Some of the notifications considered are the trace data sent for *reaching the error handler*, *new event consumed*, *new object created* and *object destroyed*. The trace data frame format for these notifications are shown in Table 5.2 and Figure 5.7.

Table 5.2: Trace data frame format for target monitor notifications

| **Notification** | **Length** | **Command_Id** | **Parameters** | **Total frame length** (bytes) |
|---|---|---|---|---|
| New event consumed | 21** | 9 | Refer Figure 5.7 | 23 |
| New object created | 8 | 7 | Refer Figure 5.7 | 10 |
| Object destroyed | 4 | 6 | Refer Figure 5.7 | 6 |
| Timeout event | 13 | 8 | Refer Figure 5.7 | 15 |
| Error handler reached | 0 | 5 | No parameters | 2 |
| Inject event (from host) | 16** | 10 | Refer Figure 5.7 | 18 |

** Length is variable based on the type and number of parameters for the events. Length of 21 (for *event consumed notification*) corresponds to an event with one integer parameter (4 bytes). Similarly, length of 16 (for inject event) corresponds to an event with one integer parameter (4 bytes).

Note that an *event consumed notification* can be of two types, i.e., it can indicate an event sent from one object to another object (e.g. *Controller* object injecting an event *evToggle(LedNr)* to *LED* object) or a timeout event for a given object (e.g. for every timeout/elapsed interval of time, say (for every) 300 ms, an event *evToggle()* is injected to the LED object by the RTOS).

For the various notifications, each trace data frame is identified by an unique identifier, which is one byte in length. Identifiers 9, 8, 7, 6 and 5 indicate *new event consumed*, *time out event consumed*, *new object created*, *object destroyed* and *error handler reached* notifications respectively (Table 5.2). These notifications are sent from the target to the host computer, whenever the respective action takes place (e.g. event consumption, object creation, etc) in the target.

The *inject event notification* is sent by the host (target debugger) to the target monitor. This notification is used to inject test input data (i.e., an event) to the embedded system. This notification has a command id of 10 (as the identifier) as shown in Table 5.2.

The following provides an outline of the notifications introduced above. The two main notifications used by the test framework approach, proposed in this thesis, namely *event consumed notification* (from target to host) and *inject event notification* (from host to target) are elaborated in the following.

#### Event consumed notification (from the target to the host)

When an event is generated and consumed in the embedded application software, this is handled by the RTOS (OORTX-RXF in the prototype). At this point, an event consumed notification is sent to the host computer, by the target monitor. For example, after an event is processed and dispatched to its respective receiver in the embedded system, the *Monitor_sendEventData(unsigned*

*int* pEventData)* function in *Monitor.c* is invoked by the RTOS with the respective event data (e.g. *int* pEventData*). The function *Monitor_sendEvent()* in turn invokes the respective operation in *MonitorIO.c* (e.g. *Monitor_IO_PutByte (int ch)*) to send the trace data to the host computer via the debug communication interface. As discussed in the previous section, depending on the debug interface the respective implementation of *MonitorIO.c* is invoked. An illustration of the steps involved in sending event consumed notification using the monitoring routine for the EIA-232 interface is provided in section 5.1.2.d.

Consider an example, wherein an event *evStatusMsg()* with parameter *144* is sent from the test environment *TestEnv* to an instance of the *PreDecoder* class. The event consumed notification (i.e., the trace data) from the target corresponding to this event execution on the target comprises of the event (id), its source, destination, current time, current state of the destination object and the parameters for the event (if any). An example of a new event consumed notification using the pre-defined frame format is already discussed in section 3.3. For the aforementioned example, the trace data comprises of the following data, <*0x00000e3d 0x01 0x0000183f 0x00405010 0x07 0x90*>. Note that these values are mapped to their respective high-level/design-level names by the target debugger using the XML file. For the above example, the mapped/decoded test result from the target debugger is <*evStatusMsg TestEnv itsPreDecoder1 230 CMD_Received 144*>. The event consumed notification is the most often used by the target monitor for a given embedded software application, in the proposed approach. This also has the longest frame size (23 bytes-corresponds to an event with one parameter) in the prototype (Figure 5.7).

On the other hand, for a time out event consumption at the target the trace data comprises of an event (id), its destination, current time and state of the destination object after event consumption. In the case of time out event (e.g. for elapsed interval of time, say 300 ms, an event *evToggleLED()* is injected at the LED object by the RTOS), the source and the destination are the same. Also, in the prototype evaluation there are no parameters for the time out events. Hence, the total length of the frame indicating a time out event at the target is approximately 13 bytes (Figure 5.7).

**Inject event notification from the host**

The test automation approach proposed in this thesis enables model-based test case execution at the host computer, whereas, only injecting the test input data/test stimuli (i.e., an event) to the embedded system (challenges (2) and (3)). Therefore, the target monitor proposed in this thesis, is also used to inject events (i.e., test input data) to the target (which executes the RTOS and the application software).

The event to be injected by the target monitor is sent by the host computer, in a pre-defined format (section 3.3). In the prototype implementation, the parameters for the test input data are encapsulated in the protocol-like format as seen in Figure 5.7. The event can be sent with or without a parameter. For example, 14 bytes of data is required for injecting an event without any parameter, 18 bytes for injecting an event with one *integer* parameter (4 bytes), 15 bytes of data for injecting with one *character* (1 byte) parameter and so on.

The event to be injected, its destination, source and event parameters are sent by the host to the target monitor. The target monitor decodes this frame and injects the respective event in the RTOS framework. This option is used by the test framework (proposed in this thesis) to send the test stimuli from the host computer to the embedded system. Once the test stimuli (injected event)

is executed at the embedded system, the event consumed notification sent by the target monitor to the host computer denotes the results for the test stimuli execution (in this case, the injected event).

For example, in order to inject an event *evStatusMsg()* with parameter *144* from the test environment *TestEnv* to an instance of the *PreDecoder* class, the inject event notification comprises of the following data string, *<evStatusMsg TestEnv itsPreDecoder1 144>*. The target debugger maps the high-level/design-level names in this data string and injects the corresponding data (with object addresses) to the embedded system. Thus, the inject event data sent from the target debugger to the embedded system corresponding to the above example is *<0x00000e3d 0x01 0x0000183f 0x90>* (refer XML file shown in Figure 5.3).

**Other notifications (from the target to the host)**

Notifications for *new object created*, *object destroyed* and *error handler reached* are some other notifications sent by the target monitor to the host as seen in Figure 5.7 and Table 5.2. For the *new object created* notification, the *create function id* and the *new object created* are the parameters for the trace data. Thus, the parameter field length is 8 bytes and the total length of the trace data is 10 bytes (Figure 5.7). The *object destroyed* notification comprises of the *destroyed object id* in its parameters. Thus the trace data is 4 bytes in length and the total length of the frame is 6 bytes. When an error handler is reached in the embedded software application, the target monitor sends a notification without any parameters. Thus, the length of the trace data to denote that an error handler is reached in the embedded software application is 2 bytes.

#### 5.1.2.d Experimental setup for the prototype evaluation of the target monitor

An experimental setup for the prototype evaluation discussed in this section is shown in Table 5.3 and illustrated in Figure 5.8.

Table 5.3: Experimental setup

| Evaluation board | MCB2140 | MCB1700 |
|---|---|---|
| Microcontroller family | ARM-7 | Cortex-M3 |
| Max.clock frequency | 60MHz | 100MHz |
| On chip RAM size | 32K | 64K |
| On chip Flash (ROM) | 512K | 512K |

The criteria for the choice of the embedded system/evaluation boards (for this thesis) are based on their ability to support an experimental evaluation on advanced microcontroller architectures (e.g. ARM-7, Cortex-M3) and support for debug interfaces (e.g. EIA-232/serial-generic interface, JTAG-based). Note that the EIA-232 is a well-known industry standard (serial interface) [5]. The JTAG standard [52] is also a widely used interface for debugging and testing of embedded systems.

Thus, for the prototype evaluation of the target monitor, experiments are carried out in two evaluation boards such as MCB2140 [70] and MCB1700 [22] (Figure 5.8). These evaluation boards allow users to create and test programs for the advanced ARM-7 or Cortex-M3 based architectures respectively. However, similar evaluation boards from various vendors [102] [22] can be used for experimental evaluation of the proposed approach.

The MCB2140 evaluation board [70] comprises of an NXP-LPC2140 ARM family microcontroller. The MCB1700 evaluation board [69] comprises of an NXP-LPC1768 Cortex-M3 microcontoller. Both

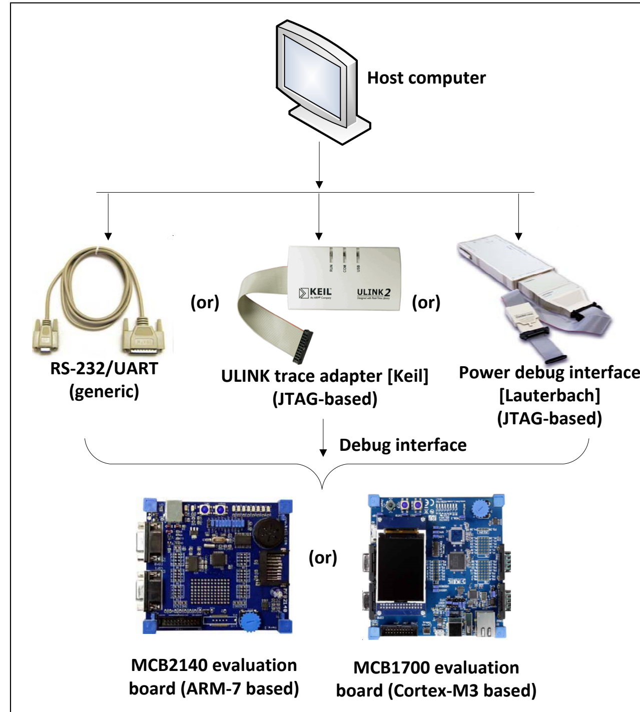



Figure 5.8: Generic and JTAG based interfaces as debug communication interfaces

the boards support EIA-232 (serial/generic) [5] and JTAG-based (industry standard) [8] debug interfaces. Debug adapters such as ULINK [110] from Keil [22] or power debug interface from Lauterbach [62] connect the target to the host computer using the respective debug interfaces (Figure 5.8). Such debug adapters are used to interpret the trace data from the JTAG interface to be used at the host computer.

Note that, among the debug interfaces evaluated, the generic EIA-232 interface is used as the debug interface for the MIDI system analyzer case study (introduced in section 4.1). The MIDI system analyzer is executed in the MCB1700 evaluation board. Hence, the development of the *MonitorIO* routine for the EIA-232 interface is elaborated in the following section (5.1.2.c).

On the other hand, the implementation specifics of the target monitor (i.e., *MonitorIO* routine) for the JTAG-based debug interfaces used in conjunction with debug and trace adapters such as ULINK [22] and power debug interface [62] are discussed in detail in, previously published works from the author such as, [48] and [46].

#### 5.1.2.d Serial/EIA-232 interface

A simple way to establish a connection between the host and the target is using a serial interface e.g. UART/EIA-232 interface. Almost every microcontroller supports a serial communication interface. The baud rate of a generic EIA-232 serial interface often has a maximum value of 115200 bit/s.

The data flow of a serial interface is event driven and often handled in Interrupt Service Routines (ISR). An interrupt occurs, whenever data is available to be read at the input buffer or to be written into the output buffer. Thus, no busy-waiting mode is required; instead the microcontroller can proceed with the application program.

In the prototype, the function *int WSTMonitor_IO_GetByte( void )* is used to read data from the host (i.e., from the input/receive buffer). An example of input data (in the proposed test framework approach) from the host is the test input data injected from the test framework. Similarly, the function *int WSTMonitor_IO_PutByte (int c)* is used to send trace data to the host over the UART interface. An example of trace data from target to host is the test result indicating an event consumed notification (Figure 5.7). The steps involved in injecting an event from the host to the target (Figure 5.9) and receiving the test result from target to host (Figure 5.10), using the target monitor implementation for the serial/EIA-232 interface, are discussed below.

**Inject event (test input data to the target)-Figure 5.9**

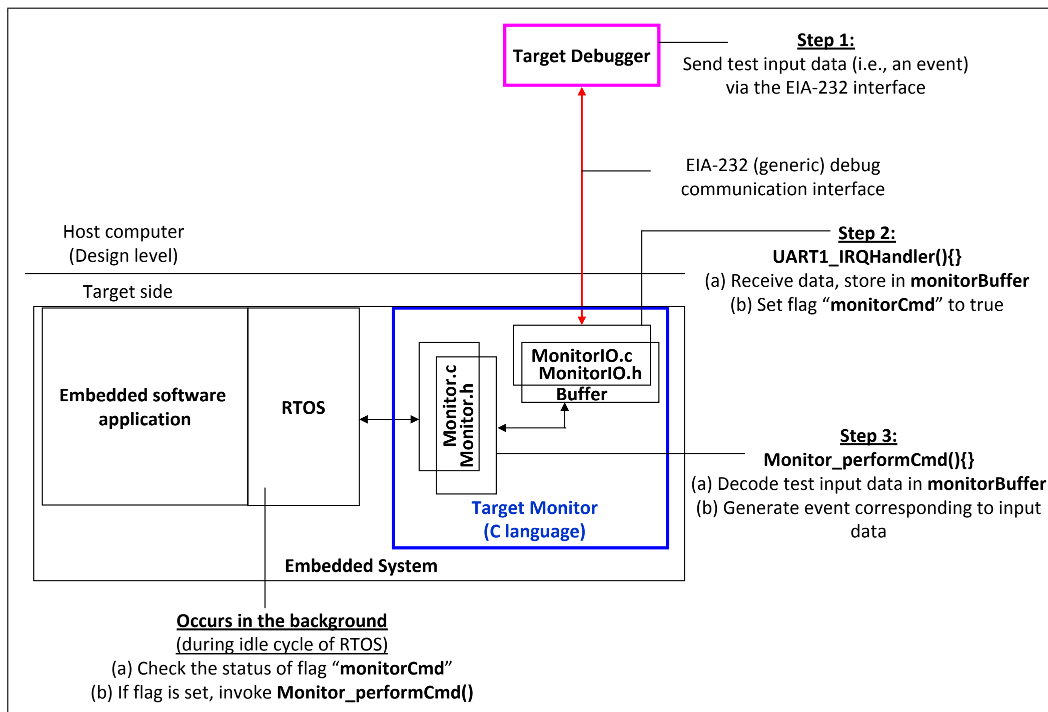



Figure 5.9: Series of steps involved in injecting an event (test input data) from the host to the target using the target monitor (EIA-232 interface)

As seen in Figure 5.9, in the first step the target debugger (host) injects an event (i.e., test input data) to the target via the EIA-232 interface. The test input data comprises of an event, its source, destination and parameters for the event (if any). The test input data (Figure 3.5) is organized in a protocol-like format, as seen in Figure 5.7.

Since the data flow of the EIA-232 interface is event driven and often handled in ISRs, an interrupt occurs indicating that data is available at the serial interface. The interrupt is handled by an ISR handler, named as *UART1_IRQHandler()* in the prototype implementation of *MonitorIO* routine (Figure 5.9). Thus, in the second step, the input data from the host is stored in a buffer named *monitorBuffer*. In this function, a flag (*monitorCmd*) is set indicating that there is some input data available from the host. Note that the size of the *monitorBuffer* is configurable in the prototype.

In the background, the RTOS checks for the status of the flag (*monitorCmd*), during its idle cycles.

If this flag is set, the RTOS invokes a function *monitor_performCmd()* in the *Monitor.c* routine to decode the incoming test input data from the host computer. The functionality of *monitor_performCmd()* is shown (in step 3) in Figure 5.9. This function, when invoked by the RTOS, decodes the incoming test input data. The test input data is available in a buffer, *monitorBuffer*. Thus, in step 3, the incoming test data in the pre-defined format (Figure 5.7) is decoded by the *monitor_performCmd()* (available in *Monitor.c* routine). Then the corresponding event (w.r.t to the incoming test data) is generated in the *monitor_performCmd()* function.

### Trace data to host (e.g. event consumed notification) - Figure 5.10

Consider an example of sending an event consumed notification (i.e., test result) to the host computer using the target monitor implementation for the EIA-232 interface. Note that the execution time in the target monitor for sending an event consumed notification (i.e., test result) to the host is split into two parts. The first part is to write the data into the send buffer (e.g. using *int WSTMonitor_IO_PutByte (int c)*). The second part is the ISR of the UART interface which sends the data to the host. The series of steps involved in sending the trace data from the target to the host using the target monitor prototype implementation for the EIA-232 interface is shown in Figure 5.10.

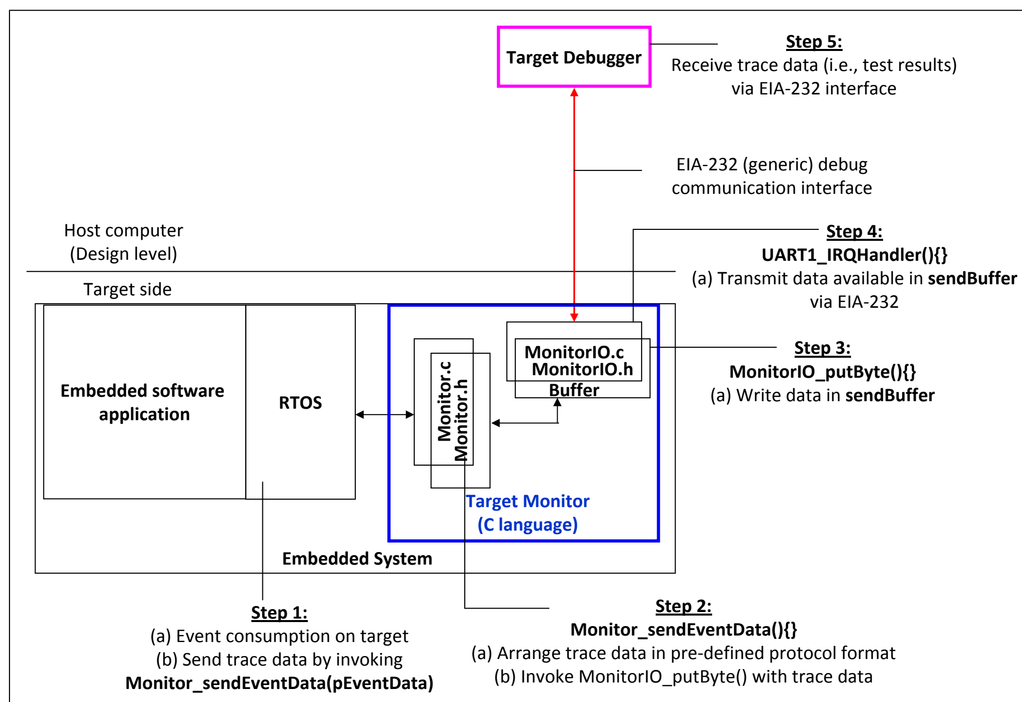



Figure 5.10: Series of steps involved in sending the trace data (e.g. test result) from the target to the host using target monitor (EIA-232 interface)

As seen in Figure 5.10 (step 1), once an event is consumed on the target, trace data about the event consumption can be sent by invoking a function *Monitor_sendEventData(pEventData)* in the *Monitor* routine of the target monitor. The trace data is now available in the pre-defined format shown in Figure 3.6. This trace data has to be encapsulated in a protocol-like format, so that it can be decoded at the host (by the target debugger). Hence, in the next step (i.e, step 2 in Figure 5.10), the trace

data is arranged in the protocol-like format for event consumed notification (Figure 5.7). The trace data is then sent for processing at the next step by invoking a function *MonitorIO_putByte()*, in the *MonitorIO* routine.

The function *MonitorIO_putByte()* (step 3 in Figure 5.10) writes the trace data in a buffer named *sendBuffer*. Note that the size of this buffer is configurable. As mentioned earlier, the data flow of the EIA-232 interface is event driven and often handled in ISRs. Therefore, between steps 3 and 4 in Figure 5.10 an interrupt occurs indicating that data is available at the *sendBuffer* (UART interface). Thus, in step 4 the ISR handler for the EIA-232 interface transmits the data available at the *sendBuffer*. Now the test result (i.e., event consumed notification) is available for further processing at the target debugger (host). This is indicated as step 5 in Figure 5.10.

Note that, if too many new events appear in a short time interval (in the target), it could result in an overflow condition. This may arise due to the data rate configured for the EIA-232 interface and also the size of the *sendBuffer*. In order to overcome this, the user could configure for discarding the overflowing event consumed notification by the target monitor. This means that the discarded notifications will not be sent to the host and the host is only notified about the overflow condition (e.g. by using *Monitor_sendBufferReset(void)* function-Table 5.1). On the other hand, another configuration option for the user is to allow the sending function (in the target monitor) to block until space is free in the buffer. However, this would result in increasing the execution time in the target monitor.

### 5.1.3 Performance metrics

In this section, performance metrics pertaining to the prototype implementation of the target monitor (discussed above) are presented. Since the proposed runtime monitoring approach is expected to be time and memory aware (challenge (3)), the time spent in the monitor routine and the memory requirement of the target monitor and for sending the various notifications are discussed. These performance metrics are obtained by measurement, based on the prototype implementation of the target monitor. The performance metrics discussed in this section are recalled during the empirical evaluation of the parameters in the complexity analysis in section 5.3.

#### 5.1.3.a Target monitor size

The target monitor memory footprint is obtained from the data available in the *linker map file* (i.e., the size of the object files involving the monitoring functionality-ROM and the RAM values). The *linker map file* is generated during the compilation of the source files corresponding to the chosen debug interface.

Table 5.4: Target monitor memory requirement for different debug interfaces (Cortex-M3)

| Interface | RAM (bytes) | ROM (bytes) |
|---|---|---|
| *Generic-EIA-232 | 112 | 1290 |
| JTAG-Keil | 84 | 1194 |
| JTAG-Lauterbach | 84 | 1396 |

*The size of the *monitorBuffer* and the size of the *sendBuffer* (for this interface) are not included in this measurement. An additional 12 bytes (RAM) per class is required for sending the trace data about the created object, i.e., initial constructor creation.

The target monitor size for the three debug interfaces (for the Cortex-M3 architecture) is shown in Table 5.4. Based on the functionality of the generic EIA-232 interface, discussed in the previous section, it is clear that there are two buffers namely *monitorBuffer* and *sendBuffer* in the prototype implementation of the target monitor (for the EIA-232 interface). Note that the size of the monitorBuffer and the *sendBuffer* are set to zero during the measurement, available in Table 5.4. As mentioned earlier, detailed implementation specifics of the *MonitorIO* routine for the JTAG-based interfaces (using debug adapters from Keil [22] and Lauterbach [62]) are available in, already published work by the author, such as [48] and [46]. However, in this section the target monitor memory footprint corresponding to the JTAG-based interfaces are also provided for a comparative study.

In the proposed approach, once the test cases are specified in the automatically generated test framework, the test cases (i.e., the test harness) are executed on the host computer. Then, only the test input test data (i.e., in the form of events) is injected to the embedded system with the aid of the target monitor. Thus, the only component necessary on the embedded system for executing the model-based test cases using the proposed approach is the target monitor. Another major advantage is that the target monitor code bundled along with a given application source code, does not vary based on the model-based test cases specified in the test framework. In other words, the target monitor is independent of the test cases (their number and complexity) to be executed on the target. In addition, the size of the target monitor routine in the embedded system is measurable and known beforehand, and is approximately 1 KiByte (for all three debug interfaces). With this total memory requirement, the monitor code can be accommodated in resource constrained embedded systems and it can remain in the final production code.

Thus, the proposed target monitor approach is memory aware and has a minimal memory requirement on the target. Therefore, the proposed approach is a strong candidate for the applicability of minimally intrusive runtime monitoring towards executing the model-based test cases in the embedded system (challenge (3)).

#### 5.1.3.b Time spent in monitor routine

The two main functionalities of the target monitor are,

- Insert the test input data to the embedded system
- Send notifications about the target behavior (e.g. event consumed notification) to the host

Based on these functionalities, the notifications for the target monitor routine were discussed in section 5.1.2.c. Among these notifications, the event consumed notification to the host computer (i.e., trace data about test result), has the longest frame size. The event consumed notification requires 17 bytes without any parameter for the event, 21 bytes with one integer parameter for the event and so on (Figure 5.7 and Table 5.2).

Consider the time spent in the monitor routine for sending an event consumed notification comprising of 21 bytes of data. This example is taken into consideration since, the events considered for the MIDI system analyzer example comprises of one integer parameter, thereby requiring 21 bytes for the trace data. The time spent in the monitor routine for sending this notification using the target monitor (in the prototype) is shown in Table 5.5. These values are measured using an oscilloscope [101]. Consider the total time spent in the monitor routine for sending the event consumed notification

Table 5.5: Time spent in the monitor routine for an event consumed notification (Cortex-M3)

| Interface | Time in monitor ($\mu$s) |
|---|---|
| Generic-EIA-232 | 74.52 |
| JTAG-Keil | 265 |
| JTAG-Lauterbach | 16.5 |

using the EIA-232 interface. This value is determined as follows. As discussed in section 5.1.2.d, the data flow of the serial interface is event driven and handled in ISRs. An interrupt occurs whenever data is available to be read at the input buffer or to be written into the output buffer. Thus, no busy-waiting mode is required; instead the microcontroller can proceed with the application program. Hence the time to send a notification using the EIA-232 interface is split into two parts. The first part is the time spent in monitor routine to write the data into the *sendBuffer*, which is 23 $\mu$s. The second part is the time spent in the interrupt routine of the EIA-232 interface which sends the data to the host. This value is calculated (from individual measurements) as 51.52 $\mu$s (enter/exit interrupt routine: 5.52 $\mu$s and time spent in ISR to send 23 bytes=2.00 $\mu$s*23). Thus the total time spent in the monitor routine (for EIA-232) is 74.52 $\mu$s.

The empirical measurements for the JTAG-based interfaces are also provided here to indicate that the time spent in the monitor routine using these interfaces, are also measurable beforehand. Detailed implementation specifics of the target monitor routine using these interfaces are available in [48], [46]. From the empirical results in Table 5.5, it is seen that the time spent in the monitor routine of the JTAG-Keil interface is significantly higher than the JTAG-Lauterbach interface. The significant difference in the time spent in the monitor routine for these two interfaces arises from the following reasoning. A polling mechanism is used to read/write data between the *MonitorIO* routine and the JTAG-Keil interface. As the monitor routine needs to poll (busy-waiting mode) for the availability of the debug interface (to write the trace data), the time spent in the monitor routine increases significantly. On the other hand, the JTAG-Lauterbach interface has an internal buffer which takes care of reading/writing the trace data. However since both these interfaces are proprietary, no further information on the internal functionality of these interfaces can be obtained.

Based on the aforementioned empirical values it is clear that the time spent in the monitor routine (in the order of $\mu$s) is measurable and known beforehand. Thus, the proposed approach is also time-aware, i.e., the time spent in the runtime monitoring routine for sending the test result is bounded and known beforehand. Irrespective of the number and complexity of the model-based test cases, the time spent in the monitor routine for injecting the test input and sending the test result (in this case an event consumed notification) is constant and measurable beforehand. This bounded time spent in the monitor routine can be included in the system design phase/accommodated in the design model by the end-user. Therefore, the proposed approach is a strong candidate for the applicability of less/minimally intrusive runtime monitoring towards executing the model-based test cases in the embedded system (challenge (3)).

This is unlike the existing approaches for monitoring/executing test cases, which require extensive (static/dynamic) source code instrumentation [45] [43]. Such techniques can result in unbounded delay, thereby introducing unbounded overhead in the embedded software application under consideration.

## 5.2 Executing model-based test cases in the embedded system

In the proposed integrated model-based approach, given a chosen SUT and the system design model, a test framework generation algorithm automatically generates the necessary artifacts (i.e., the test framework) for executing the model-based test cases in the embedded system (challenges (1), (2) and (3)). Then, the model-based test cases (e.g. UML sequence diagram test cases) are specified manually in the automatically generated test framework. The test framework in turn aids the model-based test case execution process.

UML sequence diagram-based test cases can be considered as a first choice (among alternatives for a specification of model-based test cases) to verify the interactions between the several components for a given embedded software application scenario. A (brief) background on UML sequence diagrams is available in chapter 2 (section 2.1).

In this section, executing UML sequence diagram-based test cases using the automatically generated test framework (proposed approach) for the MIDI system analyzer case study is elaborated. In section 4.1 the MIDI system analyzer embedded software application scenario is introduced. Also, the design model for the MIDI system analyzer and the automatically generated test framework for the chosen SUT (*PreDecoder* class) were discussed in detail. This section is organized as follows.

An outline of the experimental setup for the MIDI system analyzer case study is provided in section 5.2.1 (challenge (4)). Examples of UML sequence diagram test cases specified manually in the automatically generated test framework for the MIDI system analyzer case study are discussed in section 5.2.2. Then the steps involved in executing these test cases using the automatically generated test framework are elaborated in section 5.2.3 (challenges (1), (2), (3) and (4)).

### 5.2.1 Experimental setup for the MIDI system analyzer

The goal of the MIDI system analyzer case study is to develop and test a MIDI system analyzer embedded software application using the proposed integrated model-based approach and test framework. By this, challenge (4) which deals with the evaluation of the proposed approach in a real-life embedded software engineering project is envisioned to be addressed.

Consider that the MIDI system analyzer is implemented as an embedded software application using the MDD approach shown in Figure 3.1 (left side). The main functionality envisaged for the MIDI system analyzer application running on the embedded system is to interpret the incoming MIDI signals (e.g. from a key press/key release action on a musical keyboard) and generate the corresponding note, in real time.

An experimental setup for the MIDI system analyzer case study is shown in Figure 5.11. In the prototype and experimental evaluation, the MIDI system analyzer embedded software application is executed on an embedded evaluation board, MCB1700 [69]. A MIDI keyboard is connected to the evaluation board via a MIDI interface (Figure 5.11). The incoming MIDI signals (e.g. from a key press) are conveyed as bytes (e.g. data representing key press message), to the MIDI system analyzer application, by the MIDI interface. The MIDI messages are envisaged to be processed by the embedded software application (i.e., the MIDI system analyzer) executed on the evaluation board. The envisaged usage/functionality of the MIDI system analyzer and the experimental setup for executing the model-based test cases are highlighted using different line formatting in Figure 5.11. Whereas the MIDI system analyzer embedded software application is developed on a host computer (during

the MDD phase), the developed embedded software is deployed on the target using a deploy interface (Figure 5.11). The MIDI system analyzer embedded software application developed so far is subject to quality assurance using the test framework proposed in this thesis.

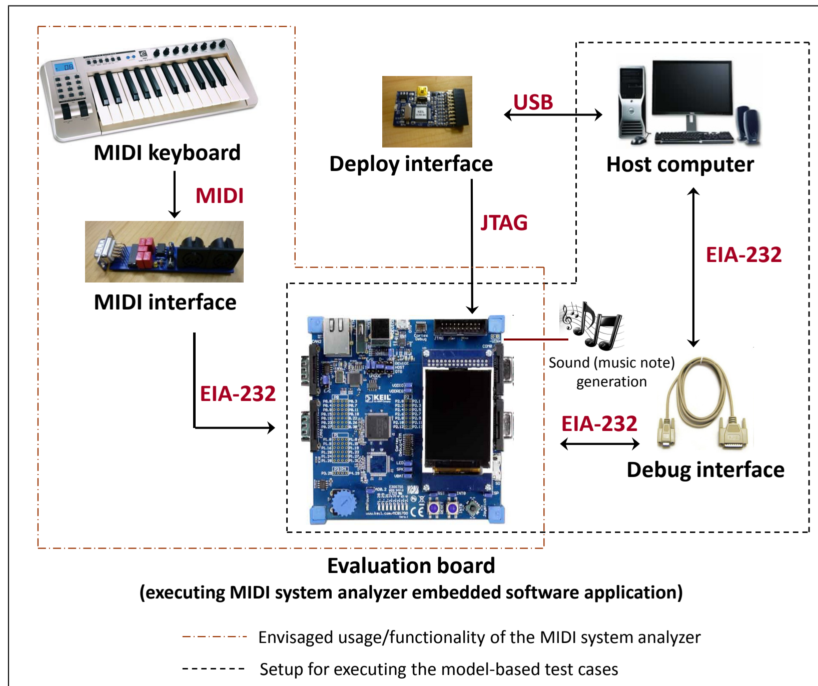



Figure 5.11: MIDI system overview and experimental setup

### MBT phase/executing model-based test cases

In the proposed approach, during the MBT phase, given a chosen SUT and the system design model (from MDD phase), a test framework is generated for executing the model-based test cases in the embedded system. The test framework makes use of a target debugger (on the host) and a target monitor (on the embedded system) for executing the model-based test cases in the embedded system. The communication between the target debugger and the target monitor takes place via a debug interface. In the MIDI system analyzer case study, a generic EIA-232 interface is used as the debug communication interface as seen in Figure 5.11. It may be noted that in certain cases a deploy interface can be used as the debug interface (e.g. JTAG based interfaces).

During the MBT phase, the model-based test cases are specified manually in the automatically generated test framework. Then, the test case execution process is carried out (automatically) with the aid of the test framework on the host. Whereas the test cases are executed on the host computer, only the test input data (e.g. an event) is injected to the target monitor via the debug interface (i.e., the EIA-232 interface in the MIDI system analyzer example). Corresponding to the test input data, the test results (e.g. event consumed notification) from the target are sent (via the corresponding *MonitorIO* routine) to the host using the same debug interface. The aforementioned setup for executing the model-based test cases using the proposed approach is shown using a different line formatting in Figure 5.11. In the following section 5.2.2, examples pertaining to model-based test case specification for the MIDI system analyzer case study are discussed.

### 5.2.2 Test case specification

The main scope of this thesis is a proposal for a test automation approach which enables model-based test case execution on the embedded system. The test cases are specified manually in the automatically generated test framework (using the proposed test automation approach). On the other hand, model-based test cases automatically generated (e.g. using MBT tools [64], test generation techniques [19], [9]) can also be included in the automatically generated test framework for a chosen SUT. Therefore this section provides only some examples for the specification of test cases (e.g. UML sequence diagram). The examples discussed in this section should be treated as one among the several alternatives for the specification of model-based test cases in the automatically generated test framework. An exhaustive description of model-based test cases (their types, specification, etc) that may be used in conjunction with the automatically generated test framework is not provided in this section.

As indicated earlier, UML sequence diagram-based test cases can be considered as a first choice among the widely used alternatives, for specification of sequence of interaction among various objects, in embedded software application scenarios. Whereas test cases based on UML diagram types such as state charts can be specified in the automatically generated test framework, four examples based on UML sequence diagram test cases are provided in this section. These example test cases illustrate a subset of different possibilities (with increasing complexity) for test specification using UML sequence diagrams w.r.t the proposed test framework.

#### 5.2.2.a Example 1.1 (Figure 5.12)

Consider an example (Figure 5.12) of an UML sequence diagram test case specified manually in the automatically generated test framework for the chosen SUT (i.e., the *PreDecoder* class). The test case shown in Figure 5.12 comprises of three objects and their lifelines. The objects are the two test component instances namely *TCon_PreDecoder_proxy.itsTestDriver_of_PreDecoder_proxy*, *TCon_PreDecoder-_proxy.itsMIDI_Interpreter_proxy* and an instance of the SUT *TCon_PreDecoder_proxy.itsPreDecoder_p-roxy*.

These test component instances are automatically generated by the test framework generation algorithm based on the chosen SUT and its corresponding design model. With respect to the expected functionality of the test case (specified in the test framework), these test component instances can be included in the test cases. For example, since the chosen SUT (i.e., the *PreDecoder* class) has one association end (with the *MIDI_Interpreter* class) in the design model (Figure 4.8), an object of the test component instance (corresponding to the association end) namely, *TCon_PreDecoder_proxy.itsMIDI_Interpreter_proxy* is specified in the test case shown in Figure 5.12.

The test driver instance (*TCon_PreDecoder_proxy.itsTestDriver_of_PreDecoder_proxy*) triggers the test case execution process by injecting the events corresponding to the input for the test case. The goal of this test case (Figure 5.12) is to verify if the respective *evValidMIDICmd()* messages are generated by the SUT on receiving the command and data bytes as external input (injected by the test driver). The test input injected by the test driver and the expected output for this test case are highlighted using dotted rectangular boxes in Figure 5.12.

The input injected by the test driver instance corresponds to a key press and a key release message. A sequence of three messages *evStatusMsg(MIDI_Cmd=144)*, *evDataByte(Byte=36)* and *ev-*

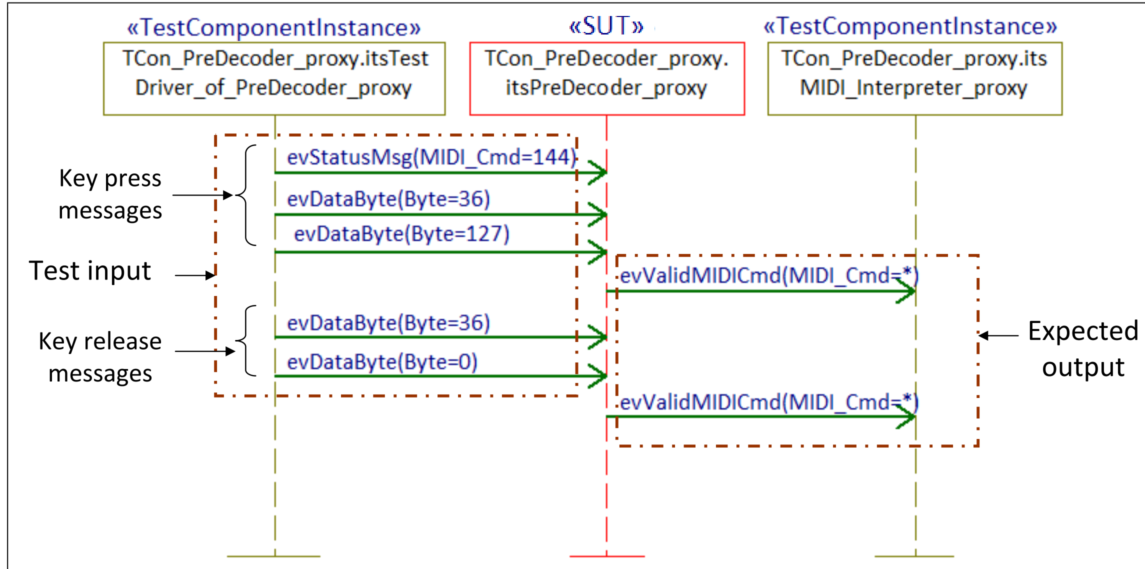



Figure 5.12: Screenshot showing an example of an UML sequence diagram test case specified manually for the MIDI system analyzer case study. Test input injected by the test driver and the expected output (messages) are highlighted within the dotted rectangle.

*DataByte(Byte=127)* as indicated in Figure 5.12 represents a key press message. In other words, this sequence of message indicates that a key is pressed on the keyboard *(evStatusMsg(MIDI_Cmd=144))* with a specific key number (*evDataByte(Byte=36)*) with an attack velocity/duration of key press indicated by *evDataByte(Byte=127)*. Thus a musical note must be played (turned on) in the MIDI system analyzer software executed on the embedded system (i.e., by generating a *evValidMIDICmd(MIDI_Cmd=*)* message), which is the expected output of the test case.

Similarly, the next two messages (test input) sent by the test driver denote that the key that was pressed, in the earlier sequence of messages, is now released. This is indicated by the sequence of messages *evDataByte(Byte=36)* and *evDataByte(Byte=0)* as seen in Figure 5.12. Thus the note that was turned on (after the earlier sequence of messages) must be turned off by the MIDI system analyzer software. It should be noted that for playing a *note* (turning on) three messages are injected. However, to turn off this corresponding note the MIDI system analyzer embedded software developed in the prototype, expects only two messages (e.g. *evDataByte(Byte=36)* and *evDataByte(Byte=0)*).

Corresponding to this sequence of messages (input for test case), the SUT needs to generate *evValidMIDICmd()* messages to the test component instance *TCon_PreDecoder_proxy.itsMIDI_Interpreter_proxy* (expected output for the test case). The parameter for the event *evValidMIDICmd(MIDI_Cmd)* has to be determined dynamically by the SUT based on the input it receives from the test driver. Thus, the goal of this test case (Figure 5.12) is to verify if the corresponding (valid) MIDI messages (represented by *evValidMIDICmd(MIDI_Cmd)*) to turn on and turn off a note are generated by an instance of the SUT *TCon_PreDecoder_proxy.itsPreDecoder_proxy*, w.r.t the input sequence of messages.

#### 5.2.2.b Example 1.2 (Figure 5.13)

The test case shown in Figure 5.12 can also be specified with verification messages for state information in the UML sequence diagram as seen in Figure 5.13. The test framework (i.e., by the communication interface component) decodes the incoming test result from the target debugger which also comprises of the state information. Based on the decoded test result, the corresponding UML sequence diagram is displayed with state information by the proxy test model component in the test framework (Figure 4.23).

In the prototype, the trace data/test result from the target always comprises of the state information as seen in the frame format of event consumed notification (Figure 3.6, Figure 5.7). Similarly, in the prototype implementation of the test framework, the state information from the test result is also decoded by the communication interface component. However, the state information is displayed in the test result (i.e., in the UML sequence diagram) only if the user specifies that it is an expected result. For example, the state information expected for the test case is not specified in the example shown in Figure 5.12. It should be noted that, the same test case example is specified with state information in the expected result (messages within the dotted rectangle), as seen in Figure 5.13. In the

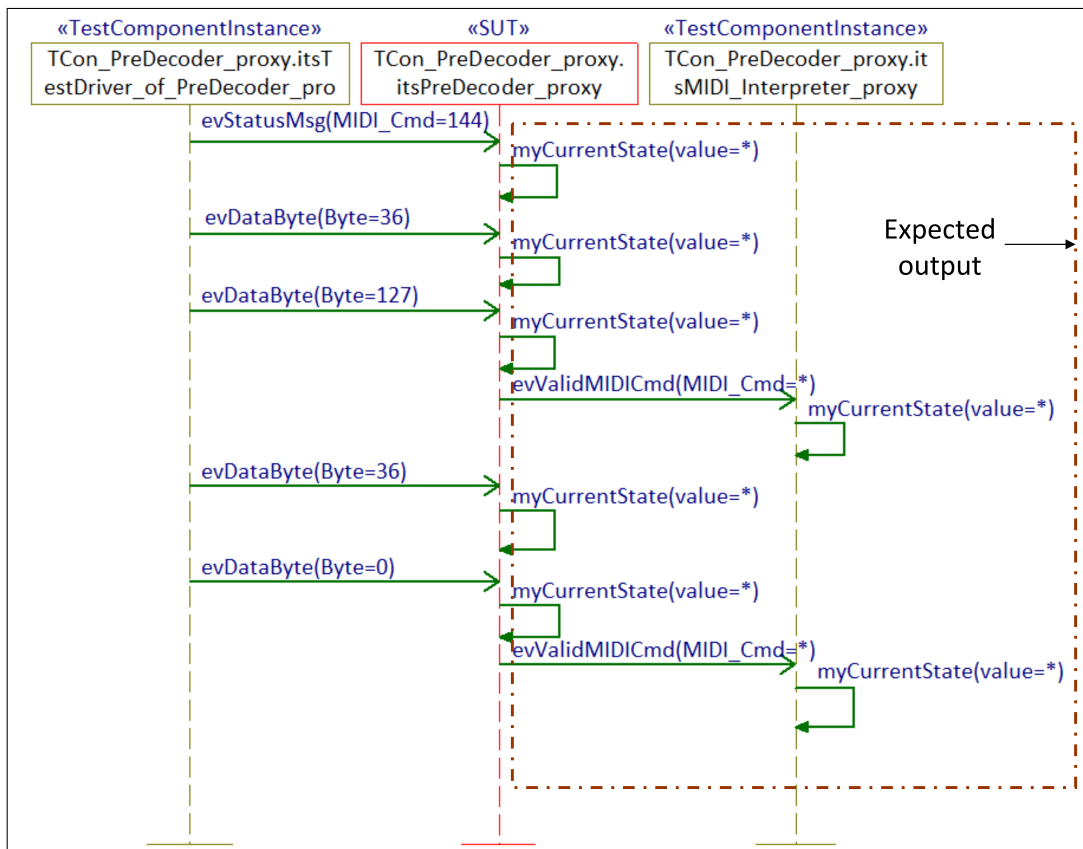



Figure 5.13: Screenshot showing an example of an UML sequence diagram test case specification with state information for participating test components

prototype implementation, the state information for the test cases is displayed by invoking a function *myCurrentState()* while decoding the trace data in the test framework (i.e., communication interface component). Therefore in the expected result for the test case, the function *myCurrentState(value)*

is used to denote the current state of the destination object (of an event). For example, on an event *evStatusMsg(MIDI_Cmd=144)* reception, the expected state value of the SUT is specified as *myCurrentState(value=*)* in the test case shown in Figure 5.13. Note that the state information can also be included in the expected test result, with a specific value for the state of the destination object, i.e., by specifying *myCurrentState(value=CMD_Received)*. In this case, the expected state value is *CMD_Received*.

### 5.2.2.c Example 1.3 (Figure 5.14)

The goal of this test case is to demonstrate the specification of more than one sequence of input messages (i.e., key press and key release), to be executed at the target for the MIDI system analyzer case study. The test case specified in the example shown in Figure 5.14 illustrates a sequence of two key press/key release messages given as input by the test driver (*TCon_PreDecoder_proxy.itsTestDriver_of_PreDecoder_proxy*) to an instance of the SUT (*TCon_PreDecoder_proxy.itsPreDecoder_proxy*). The expected output from the SUT (corresponding to the input from the test driver) are also specified in the test case. The input message sequences in the test case and the expected output from the test case are highlighted using rectangular boxes and different line formatting in Figure 5.14. In

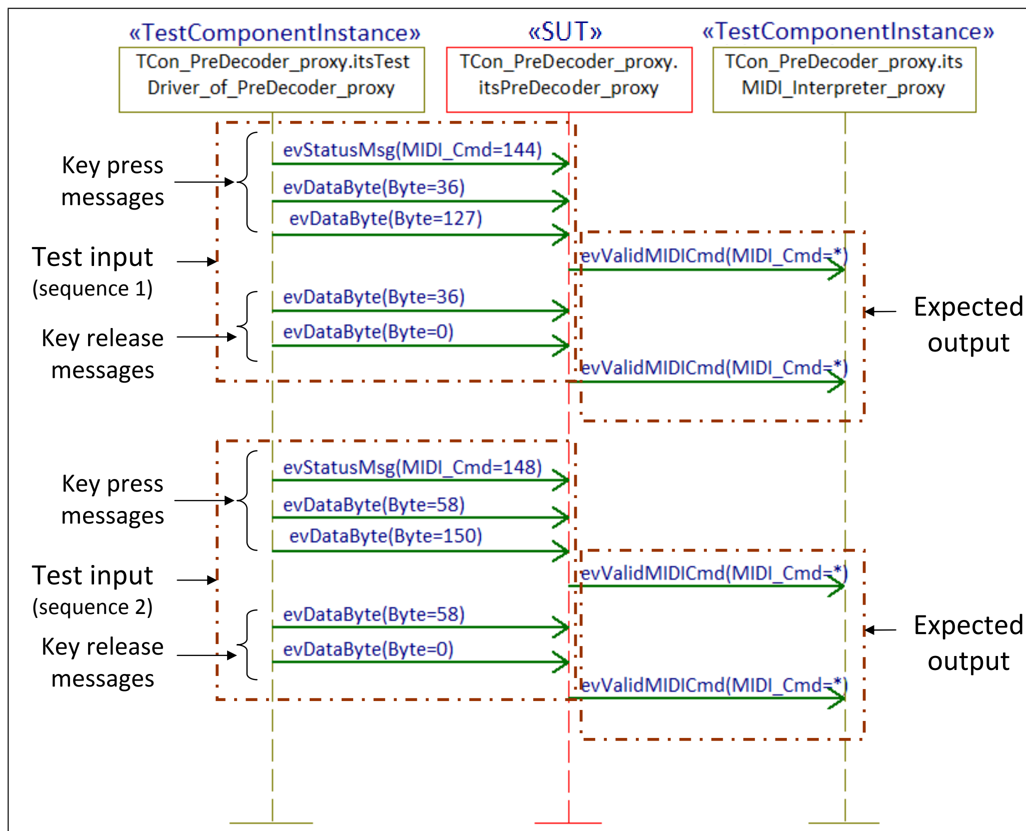



Figure 5.14: Screenshot showing an example of an UML sequence diagram test case specification with several sequences of events as input

the example shown in Figure 5.14, the input (sequence/data) from the test driver comprises of two key press/release messages which are as follows.

- Input sequence 1:
  - Key press: *evStatusMsg(MIDI_Cmd=144)*, *evDataByte(Byte=36)*, *evDataByte(Byte=127)*
  - Key release: *evDataByte(Byte=36)*, *evDataByte(Byte=0)*
- Input sequence 2:
  - Key press: *evStatusMsg(MIDI_Cmd=148)*, *evDataByte(Byte=58)*, *evDataByte(Byte=150)*
  - Key release: *evDataByte(Byte=58)*, *evDataByte(Byte=0)*

Since the input for the MIDI system analyzer example, i.e., the MIDI command messages may comprise of command bytes in the range of 128-255 (Figure 4.2), the parameter *MIDI_Cmd* for the event *evStatusMsg()* is provided with varying inputs (e.g. 144, 148) in the above input sequence (sequence 1, 2).

The expected output from the SUT (*TCon_PreDecoder_proxy.itsPreDecoder_proxy*) is the generation of an event *evValidMIDICmd(MIDI_Cmd)* corresponding to a sequence of key press/release messages (input sequence 1, 2). Thus, as mentioned above, the goal of this example is to demonstrate that more than one sequence of messages can be specified as input to the UML sequence diagram test case. The test case is then executed using the proposed test automation approach, i.e., using the automatically generated test framework for the chosen SUT.

#### 5.2.2.d Example 2 (Figure 5.15)

The goal of this example is to demonstrate the usage of the *combined fragment operator* in sequence diagrams (during test case specification and execution using the proposed approach). Combined fragments are logical groupings, represented by a rectangle, which contain the conditional structures that affect the flow of messages in UML sequence diagrams. Thus, a combined fragment defines a combination/expression of interactions by using a combined fragment/interaction operator. The interaction operator identifies the type of logic or conditional statement that defines the behavior of the combined fragment. Examples of these operators in sequence diagrams are *loop* (to define iteration), *alt* (to specify alternative sequences), *break* (to specify a breaking scenario), etc [79] [20].

In the sequence diagram test case specified in Figure 5.15, the same (input) sequence of messages are sent over several iterations (e.g. five iterations). This is possible by using a *loop* combined fragment operator in the sequence diagram test case. The operator is also specified with a guard condition to control the repetition.

In this example, a sequence of messages denoting key press and key release (injected as input by the test driver to the SUT) are iterated using a *loop* operator as seen in Figure 5.15. The iteration is controlled by a guard, which specifies the number of iterations through which a given sequence of message(s) should be executed. In the example shown in Figure 5.15, the sequence of messages are iterated for say, five times. This is specified in the test case by controlling a guard parameter *in*. This *guard* parameter is invoked as *itsTCon_PreDecoder_proxy.in*, as the parameter is declared (manually) in the test context element *TCon_PreDecoder_proxy* (automatically generated by algorithm 1).

In the test case, once the given sequence of messages injected by the test driver (during an iteration), the guard parameter is incremented. This is specified as a post-call-action wherein the guard parameter is incremented (*itsTCon_PreDecoder_proxy.in++*). After each iteration the next set of key

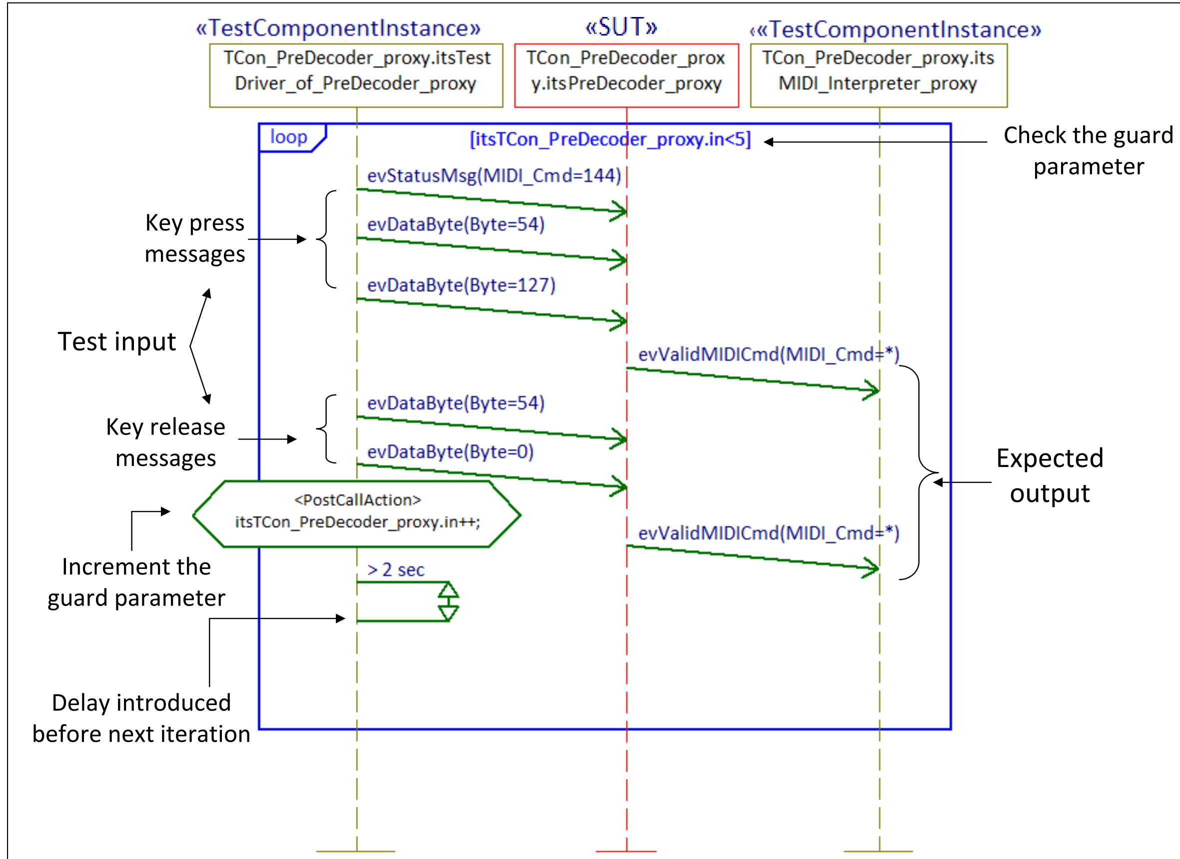



Figure 5.15: Example of an UML sequence diagram test case specification with *loop* iteration operator

press/release messages is injected as input after introducing a time delay of two seconds, as indicated in Figure 5.15. The choice of the guard parameter (iteration over five times) and the delay introduced (two seconds) can be any value chosen by the end-user/tester.

From the UML sequence diagram test case specified in Figure 5.15, it is demonstrated that non-trivial test cases (such as UML sequence diagrams with interaction operator, guards, delays) can be used (for execution) in conjunction with the proposed test framework.

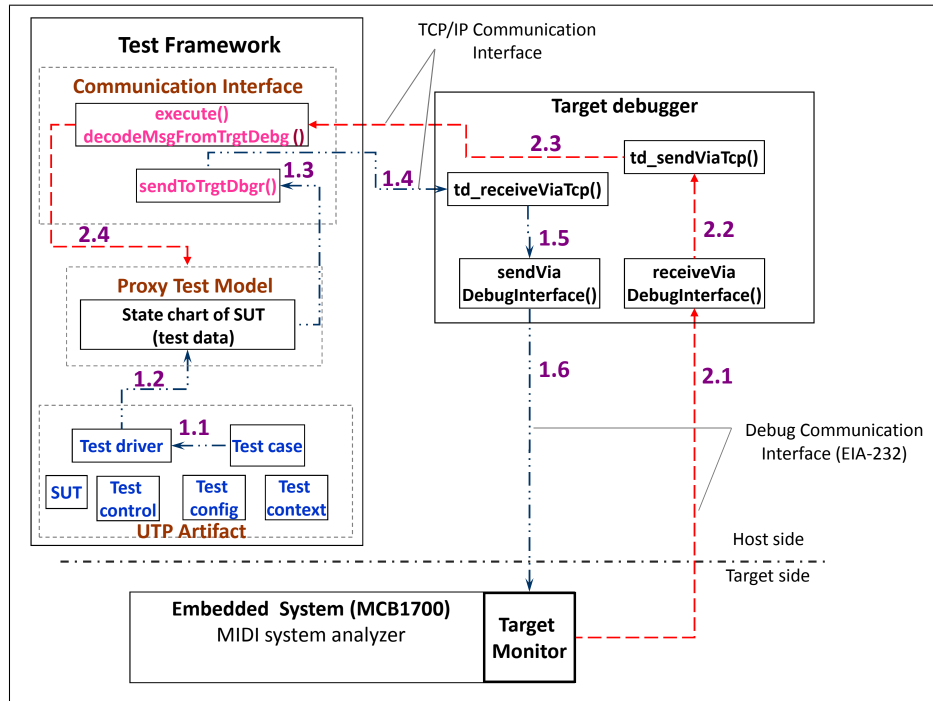
The examples discussed above are specified manually in the automatically generated test framework for the chosen SUT (*PreDecoder* class). Based on the specification of the test cases, it is clear that the test driver generated automatically by the test framework generation algorithm is used to initiate and drive the test case execution process. The several lifelines in the test cases correspond to the instance of the proxy classes generated in the test framework, corresponding to a chosen SUT.

In the prototype, the test framework is generated in a MBT/MDD tool, Rhapsody [43] by the test framework generation algorithm (i.e., algorithm 1). These model-based test cases are intended to be executed using the proposed approach and the automatically generated test framework.

### 5.2.3 Test case execution

The steps involved in executing the model-based test cases and visualizing the test results, using the proposed test framework, are discussed in this section. Consider the test case shown in Figure

5.12 which is manually specified in the automatically generated test framework for the chosen SUT (*PreDecoder* class). The series of steps involved in the test case execution using the proposed approach, for the chosen example, are discussed below.




Steps 1.1 to 1.6: executing the test cases
Steps 2.1 to 2.4: interpreting the test results

Figure 5.16: Steps involved in executing the model-based test case examples

Once the test case execution begins, the *execute()* operation in the communication interface component ($T^{Com}_{receive}$) of the test framework starts its functionality in its own thread of execution. By default, it listens to the connected TCP/IP port with the target debugger. It may be recalled that in the prototype, the communication interface component is used to send the test input data from the test framework to the target debugger and also to send the test results from the target debugger to the test framework (i.e., test framework$\xrightarrow{\text{test input data}}$target debugger and target debugger$\xrightarrow{\text{test results}}$test framework).

The test cases are executed in the embedded system using the steps illustrated in Figure 5.16. In the proposed approach, the test case execution process itself comprises of two main steps, namely.

- Executing the test cases (i.e., the test harness) in the host computer and injecting the test data alone at the embedded system. This is illustrated as steps 1.1 to 1.6 in Figure 5.16.
- Receiving, processing (decoding) and interpreting the test input data execution results from the embedded system at the host computer. This is illustrated as steps 2.1 to 2.4 in Figure 5.16.

These steps are indicated by different line formatting in Figure 5.16. The activities involved in these two steps are discussed in detail below.

#### 5.2.3.a Executing the test cases

In the first step, the test driver in the UTP artifact component of the test framework initiates the test case execution process, thereby injecting an event to trigger the SUT. This is illustrated as step 1.1 and 1.2 in Figure 5.16. Consider the test case illustrated in Figure 5.12. In this example, as a first step an object instance of the test driver, namely *TCon_PreDecoder_proxy.itsTestDriver_of_PreDecoder_proxy* injects an event *evStatusMsg(MIDI_Cmd=144)* to the SUT (*TCon_PreDecoder_proxy.itsPreDecoder_proxy*).

On receiving the event injected by the test driver, the state chart of the SUT, i.e., the *PreDecoder_proxy* class performs the respective action in the transition of the triggered event. The main functionality in the transition of the triggered event involves invoking the functions in the communication interface component with the test data corresponding to the triggered event. This is shown as step 1.3 in Figure 5.16. The state chart of the *PreDecoder_proxy* class is illustrated in Figure 4.18. As seen in this state chart, the corresponding action for the transition with event *evStatusMsg()* involves invoking the *sendToTrgtDbgr()* function with the respective test input data. The test (input) data is available in a pre-defined frame format (Figure 3.5) in the state chart of the SUT (Figure 4.18). For this example (with event *evStatusMsg(MIDI_Cmd=144)*), the test input data string is <*evStatusMsg TestEnv itsPreDecoder1 144*>. The individual string tokens of this message are delimited by a space. This test data indicates that an event *evStatusMsg()* is injected from the test environment (i.e., *TestEnv* indicating the test framework) to an instance *itsPreDecoder1* of the *PreDecoder* class. The parameter for the injected event is 144. The parameter for the event is dynamically taken as input from the specified test case (i.e., from *params→MIDI_Cmd* as seen in Figure 4.18).

Thus the test input data is conveyed to the target debugger by the communication interface component in the test framework, as seen in step 1.4. The target debugger receives this test input data and interprets it. Based on the decoded test input data, the target debugger sends the test stimuli to the embedded system via the APIs supported by the debug communication interface. These steps are shown as steps 1.5 and 1.6 in Figure 5.16. Thus the test input data is conveyed to the embedded system by the target debugger via the target monitor in the embedded system.

#### 5.2.3.b Interpreting the test results

One of the main goals of the proposed test framework approach is to eliminate the need for sending significant amount of test data (test input or test result) between the host and the embedded system (challenge (3)). In order to address this aspect, pertinent data about the target and the application software are stored in a XML file (mapping between design-level names object addresses) at the host computer. The decoding program in the target debugger maps the object addresses to design-level names and vice-versa. Then the test data is sent in a pre-defined format as illustrated in Figure 5.7. In this context, based on the steps 1.1 to 1.6 discussed above, the test data sent by the target debugger to the target monitor is decoded by the target monitor (Figure 5.16). Then the test event is executed in the embedded system. Each test input data comprises of an event (in the pre-defined format shown in Figure 5.7) to be executed corresponding to a test case. The results of the test case execution (e.g. *event consumed* notifications in Figure 5.7) corresponding to the execution of the test input are sent by the target monitor to the target debugger via the debug communication interface. This is shown as step 2.1 in Figure 5.16.

The incoming test results are received and processed by the target debugger at the host. In the MIDI system analyzer embedded software application example, the generic EIA-232 interface is used as the debug communication interface (Figure 4.1, Figure 5.10). The target debugger interprets the test data by mapping the received values to the XML file that was generated automatically at the host computer. It then conveys the decoded test data to the test framework using the corresponding communication interface component in the test framework (Figure 3.8). This is illustrated as steps 2.2 and 2.3 in Figure 5.16.

The communication interface component in the test framework receives and decodes the incoming test results and invokes the respective functions in the proxy test model. This is illustrated as step 2.4.

For the example discussed above (i.e., *evStatusMsg(MIDI_Cmd=144)*), the test result sent by the target debugger is *evStatusMsg TestEnv itsPreDecoder1 230 CMD_Received MIDI_Cmd=144*. The individual string tokens of this message are delimited by a space. This message indicates that an event *evStatusMsg()* was executed at the embedded system, the source and destination being the test environment (*TestEnv*) and an instance of the SUT (*itsPreDecoder1*) respectively. The time elapsed between the current event executed on the target and the previous event is determined by the target debugger as *230 ms*. The current state of the destination object *itsPreDecoder1* after the *evStatusMsg()* event is executed, is *CMD_Received*. The parameter for the event *evStatusMsg()* is *MIDI_Cmd=144*. Based on this decoded data the respective functions are invoked in the proxy test model to mirror the test case execution process at the embedded system.

During this step, in the background, the respective UML diagrams are drawn automatically in the test framework based on the invoked events and functions in the proxy test model (Figure 4.23). This feature is supported by the MDD/MBT tool [42], [43] used for the prototype implementation and experimental evaluation.

The above steps are repeated for all the test input/triggers (events) from the test driver to the SUT in the test case (e.g. Figure 5.12). While the test input is injected by the test case, the test results are also obtained in the background and decoded by the test framework. UML sequence diagrams denoting the decoded test data are drawn in the background by the MDD/MBT tool in which the test case is specified. The test results are obtained in real time from the target (online test execution) and visualized in the test framework (i.e., in MDD/MBT tool Rhapsody in the prototype evaluation).

#### 5.2.3.c Test results visualization

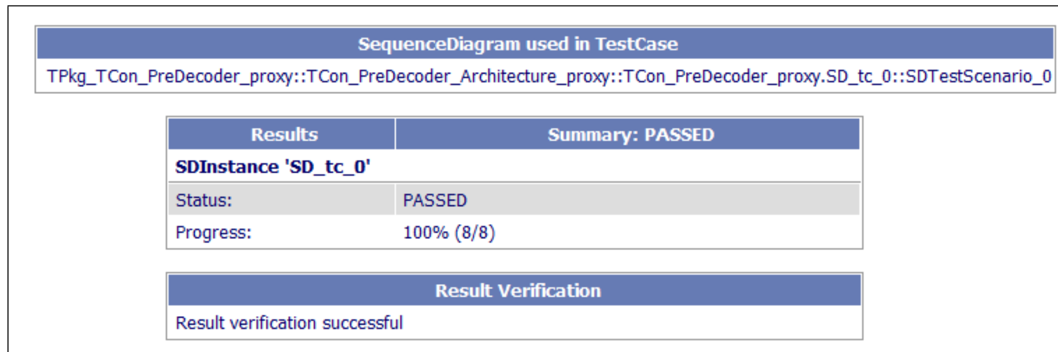
In the prototype, the test result obtained from the target is compared (in the background) with the test input specified in the test case (in the test framework) during the test case execution process. Once the test case execution process is completed, a pass/fail result for a given test case is displayed in the MBT tool. The test results can be displayed as follows:

- Summary in a .html file
- Graphical representation (e.g. UML sequence diagram) and
- Summary of the pass/fail results for the major steps in the test case execution process

Examples based on each of the aforementioned representations in the prototype implementation are discussed below.

### Summary of the test case execution results in a .html file

A summary of the test case execution results is generated as a .html file, in the prototype, as seen in Figure 5.17. This test result summary (as .html file) corresponds to the test case specified in example



| SequenceDiagram used in TestCase |
|---|
| TPkg_TCon_PreDecoder_proxy::TCon_PreDecoder_Architecture_proxy::TCon_PreDecoder_proxy.SD_tc_0::SDTestScenario_0 |

| Results | Summary: PASSED |
|---|---|
| SDInstance 'SD_tc_0' | |
| Status: | PASSED |
| Progress: | 100% (8/8) |

| Result Verification |
|---|
| Result verification successful |

Figure 5.17: Screenshot of test case execution result summary (in .html file). This result corresponds to the test case specified in Figure 5.12

1 (Figure 5.12). The number of test input data/events in the test case example 1 (Figure 5.12) is 7. The test case execution result in Figure 5.17 shows the number of events in the test case that have been successfully executed. In this example, the result is shown as 8/8 passed, as the test driver is also counted as an event in the test case execution process (tool specific).

### Graphical representation of the test results (e.g. using UML sequence diagrams)

Also, the test case execution results can be obtained as an UML sequence diagram as shown in Figure 5.18. This is similar to the graphical representation of the test case specification using UML sequence diagram illustrated, for example, in Figure 5.13.

Based on the decoded test result in the test framework, respective actions (invoking an operation, event generation, etc) are carried out in the test framework. For example, consider that a test result obtained from the target indicates than an event *evStatusMsg()* is sent from the test environment to an instance of the *PreDecoder* class. This test result is received in a pre-defined format namely, *evStatusMsg TestEnv itsPreDecoder1 230 CMD_Received MIDI_Cmd=144*. The test result is then decoded by the components of the test framework at the host computer. Corresponding to the decoded test result (for the above example), an event *evStatusMsg()* is injected from the instance of the test driver (*TCon_PreDecoder_proxy.itsTestDriver_of_PreDecoder_proxy*) to an instance of the *PreDecoder_proxy* class in the test framework. The state information of the destination object (i.e,. instance of the *PreDecoder*) on the target is changed to *CMD_Received* for the above example. Correspondingly, the state of the *PreDecoder_proxy* class is also changed to *CMD_Received* and displayed in the test result as *myCurrentState(value=CMD_Received)* (Figure 5.18).

Similarly, for the various test input sequences, the test results are displayed (for the chosen SUT) as seen in Figure 5.18. For example, the current state of (an instance of) the *PreDecoder* class after *evDataByte(Byte=36)* and *evDataByte(Byte=127)* event consumptions on the target is shown as *DataByte1* and *DataByte2* respectively (in the instance of *itsPreDecoder_proxy* class) in Figure 5.18. In other words this is indicated in the test result as *myCurrentState(value=DataByte1)*) and *myCur-*

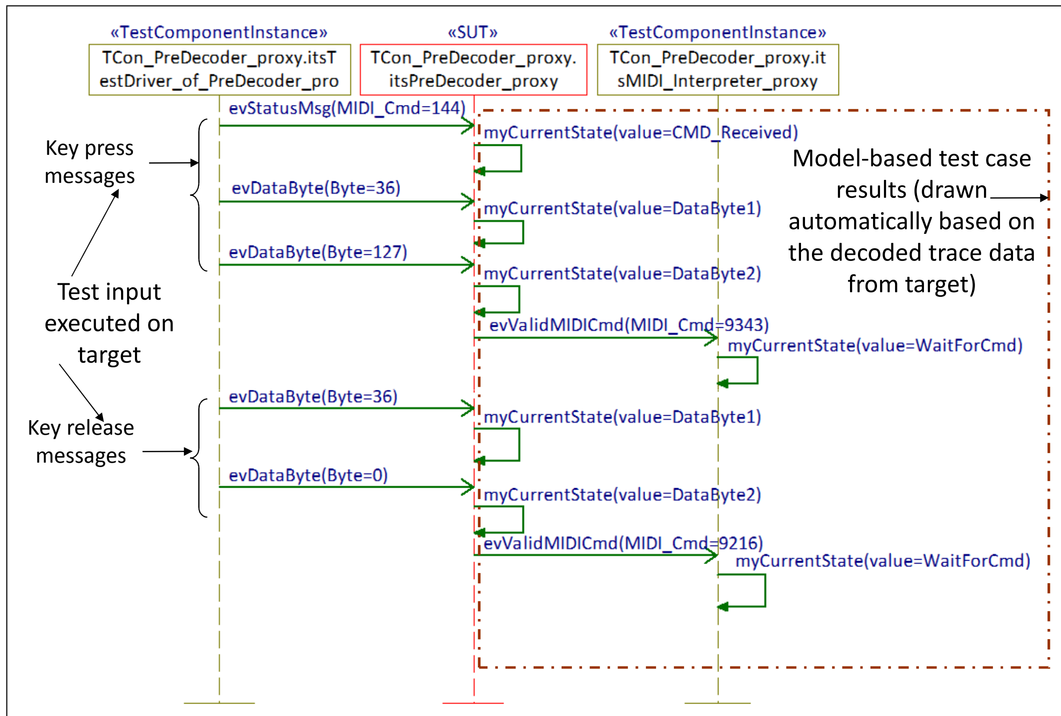



Figure 5.18: Screenshot of test case execution result with state information in the test framework. This result corresponds to the test case specified in Figure 5.13

*rentState(value=DataByte2)* in the instance of the *PreDecoder_proxy* class namely *TCon_PreDecoder_proxy.itsPreDecoder_proxy* (Figure 5.18).

Once the sequence of messages (test input) representing a key press event is received and processed by the *PreDecoder* class (in the target), it generates a valid MIDI command on the target. This test result is also decoded and shown in Figure 5.18. Thus, corresponding to an incoming test result indicating that (an instance of the) *PreDecoder* class generates an event *evValidMIDICmd(MIDI_Cmd=9343)* to an instance of the *MIDI_Interpreter* class on target, an event *evValidMIDICmd(MIDI_Cmd=9343)* is generated from an instance of the *PreDecoder_proxy* class to an instance of the *MIDI_Interpreter_proxy* class in the test result (i.e., in the automatically generated test framework). On this event consumption, the current state of the *itsMIDI_Interpreter1* object is changed to *WaitForCmd* on the target. This is also displayed in the test result as a state change of the *MIDI_Interpreter_proxy* class to *WaitForCmd* (i.e., *myCurrentState(value=WaitForCmd)*).

Thus, corresponding to the test results obtained from the target (e.g. *evAtoB()* from class *a* to class *b*), the respective operations are carried out by invoking the corresponding functions in the proxy classes (e.g. *evAtoB()* from class *a_proxy* to class *b_proxy*) of the automatically generated test framework. It can be recalled that this decoding and interpreting logic for the test framework is automatically generated by the test framework generation algorithm. Examples pertaining to the decoding and interpreting logic are discussed in section 4.4. When the expected output (e.g. Figure 5.13) in the test case is the same as the decoded test result from the target (e.g. Figure 5.18) the test case is considered as passed.

#### Summary of the pass/fail results for the major steps in the test case execution process

At the end of the test case execution process, some of the major steps of the test case execution process are displayed along with a pass/fail result, as shown in Figure 5.19 and Figure 5.20. Figure

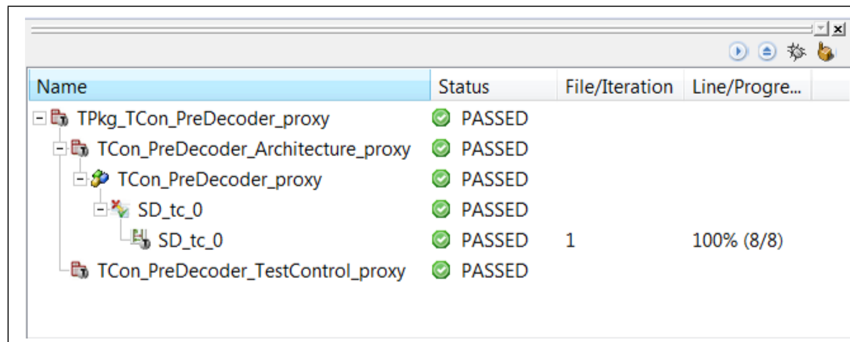



Figure 5.19: Screenshot showing the status of the steps in a test case execution. This example shows a *pass* result.

5.19 represents the pass result for the test case execution, indicating that all the steps during the test case execution have passed. On the other hand, the test case execution results in Figure 5.20

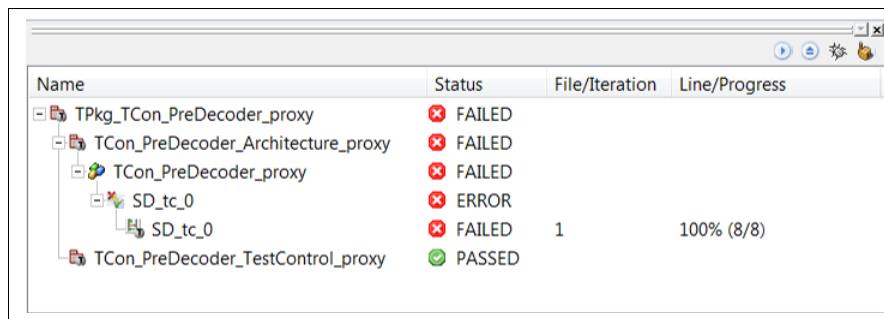



Figure 5.20: Screenshot showing the status of the steps in a test case execution. This example shows a *fail* result.

indicate that the test case has failed. However, only the control component of the test framework is displayed as passed. This is because, this component in the test framework comprises of the arbiter and scheduler which compute the test results. Thus, the results in Figure 5.20 indicate that the test case has failed (during its execution). However the control component (i.e., the arbiter and scheduler) has successfully computed a result for test case (in this case a *fail* result).

#### 5.2.3.d Verification of functional requirements at unit/module level

Test case execution typically involves the verification of the given set of requirements for a software application. In the case of an embedded software application, the requirements are often classified into functional and non-functional aspects. Therefore, the test case execution and verification for embedded software application examples often involve verification of functional requirements as well as non-functional requirements such as temporal constraints.

Verification of functional requirements at unit/module level using the proposed test framework approach is briefly discussed below. Similarly, verification of non-functional requirements such as the

temporal properties using the proposed approach is elaborated in section 5.3.2.b. These aspects are discussed w.r.t to the MIDI system analyzer example and its functional and non-functional requirements.

The requirements (both functional and non-functional aspects) of the MIDI system analyzer embedded software application are introduced in section 4.1. The examples shown in Figure 5.12, 5.13, 5.14 and 5.15 demonstrate the specification of functional test cases for the MIDI system analyzer case study. The steps involved in the (model-based) test case execution and visualization of the test results using the proposed approach are explained in the previous section.

Based on the above discussion, it is clear that the test cases specified for the MIDI system analyzer embedded software application (for the chosen SUT, i.e., *PreDecoder* class) are executed at the target without downloading the test harness in the embedded system. It is also evident that (unit/module-level) testing of functional requirements specified (at the model-level) using UML sequence diagram test cases is feasible using the proposed approach. The test results are visualized using the test framework at model-level (Figure 5.18). Also, a summary of the results is provided in a .html file (Figure 5.17) in the test framework. Thus, functional test cases specified for a chosen SUT (e.g. a class/module) can be verified automatically using the proposed test framework approach.

An overview of the steps involved in executing the test cases using the proposed approach and the screen shots illustrating the various components involved in the test case execution process are illustrated in Figure 5.21. This figure is intended to provide an overall view of the mapping of the various components (automatically generated/manually specified) to the steps involved in executing the test cases using the proposed approach.

An example involving the verification of non-functional requirements for the MIDI system analyzer example, such as verification of temporal constraints using the target debugger GUI is discussed in the section 5.3.2.b.

## 5.3 Complexity analysis and discussion

One of the main challenges of this thesis is to propose a test automation approach, for executing the model-based test cases in the embedded system, with predictable complexity measures i.e., generation and runtime overhead (challenge (3)). Towards this direction, a theoretical analysis on the complexity measures for the test framework generation algorithm and the proposed approach is discussed in detail in section 4.5 (chapter 4). The complexity analysis is provided in two perspectives namely:

- Test framework *generation* complexity using the proposed test framework generation algorithm (algorithm 1 in section 4.4.1)
- The *runtime* complexity of the proposed approach while executing the test cases in the embedded system.

In this section, empirical results based on the complexity analysis of the proposed approach and the test framework generation algorithm are presented. This empirical evaluation is envisaged to provide important insights on the overhead parameters (both during the generation of the test framework and during executing the test cases at runtime) using the proposed approach (challenge (3)).

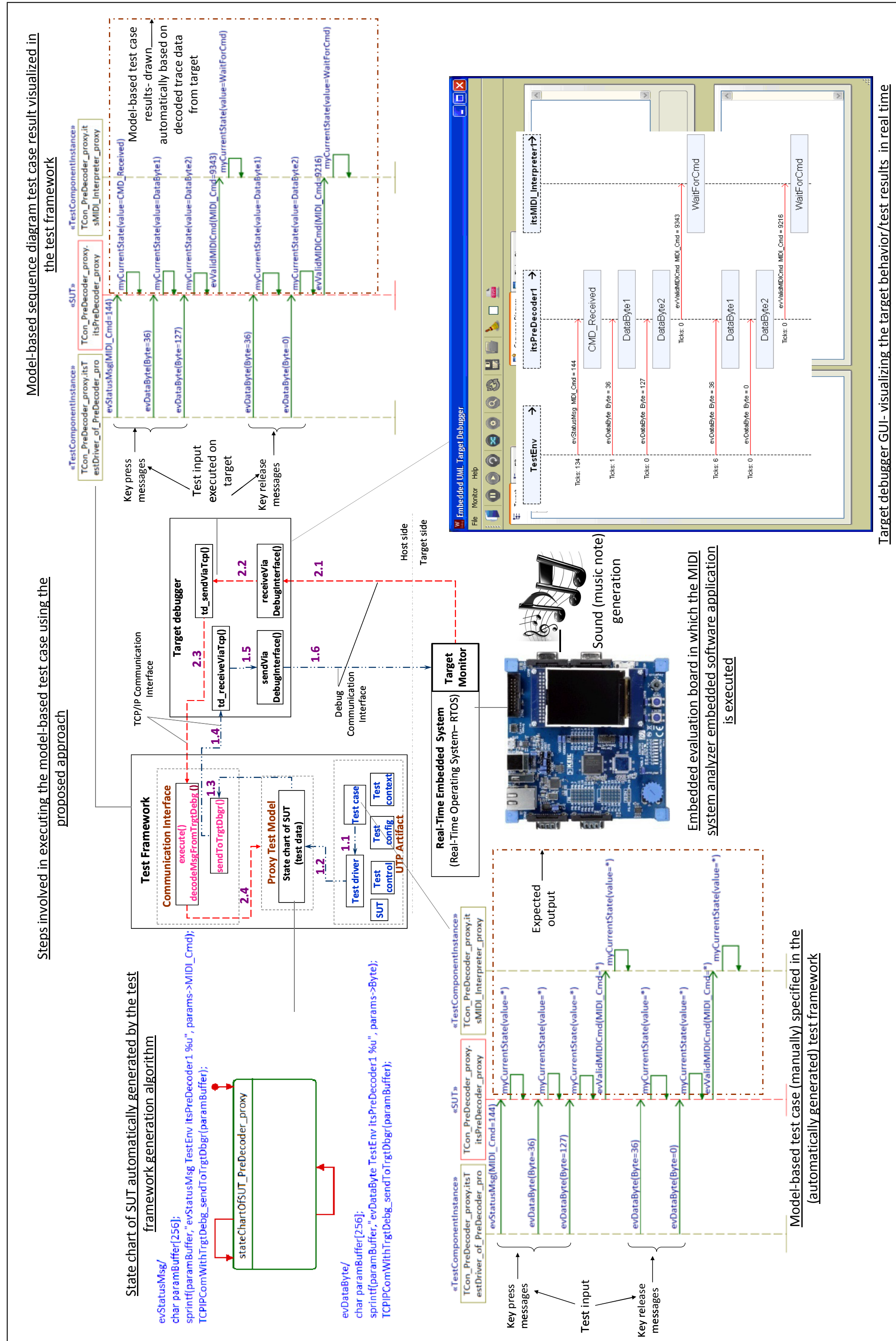



Figure 5.21: Overall view of the mapping of the various components in the test framework to the steps involved in executing the test cases using the proposed approach

### 5.3.1 Test framework generation complexity

In the prototype, the test framework generation algorithm is implemented with the aid of the APIs [88] in the MBT tool (in the programming language Java) and integrated in the MBT tool Rhapsody Test Conductor-Add on [43]. For an empirical evaluation of the generation complexity, the test framework generation algorithm (and the proposed approach) are applied to various scenarios based on the three evaluation examples (Figure 5.1) namely:

- MIDI system analyzer
- Stopwatch application
- Spark extinguishing system

The MIDI system analyzer example has already been introduced in chapter 4. A brief outline of the stopwatch application example and the spark extinguishing system example is provided below.

#### 5.3.1.a Stopwatch application example

The basic functionality envisaged for the stopwatch (embedded software) application is to start a timer on receiving a trigger event to start the stopwatch. Similarly, when an external trigger event is sent to stop the stopwatch, the timer is stopped. The time elapsed between the start and stop of the stopwatch is displayed as the output/result of the timer. Among the three example scenarios considered for experimental evaluation, the stopwatch application scenario could be considered as the least sophisticated one, because of the nature of the underlying functionality envisaged for this example.

The stopwatch embedded software application is executed on an embedded evaluation board such as MCB1700 [69]. In this case, a small display on the board shows the running stopwatch. The functionality of the stopwatch embedded software application example (implemented using the MDD phase in Figure 3.1-left side) is illustrated in Figure 5.22.

The stopwatch embedded software application is implemented such that it is started/stopped by pressing a button on the evaluation board. A button press generates an external interrupt event, which is handled by the stopwatch embedded software application. As seen in Figure 5.22, the stopwatch application waits for an external interrupt. A button press on the evaluation board (embedded system), on which the application is executed, results in the generation of two events, namely *evPress()* and *evRelease()*. When triggered by the external input combination of *evPress()* immediately followed by *evRelease()*, an event *evStartStop()* to start or stop the stopwatch is expected to be generated (Figure 5.22). Once triggered by *evStartStop()* for the first time, the *Stopwatch* class starts a timer. When the *Stopwatch* class receives *evStartStop()* again, it stops the internal timer. The time elapsed in this interval is displayed in the external display in the evaluation board.

The stopwatch is reset by a long press of the external input button on the evaluation board, i.e., if the button on the evaluation board is pressed for more than three seconds, it is considered by the embedded software as a reset for the stopwatch application. This is interpreted as an *evPress()* event reception, followed by a pause of three seconds (can be any value chosen by the developer) and then an *evRelease()* event reception. These series of events results in an *evReset()* event to be generated (Figure 5.22). The functionality to start, stop and reset the stopwatch are handled in

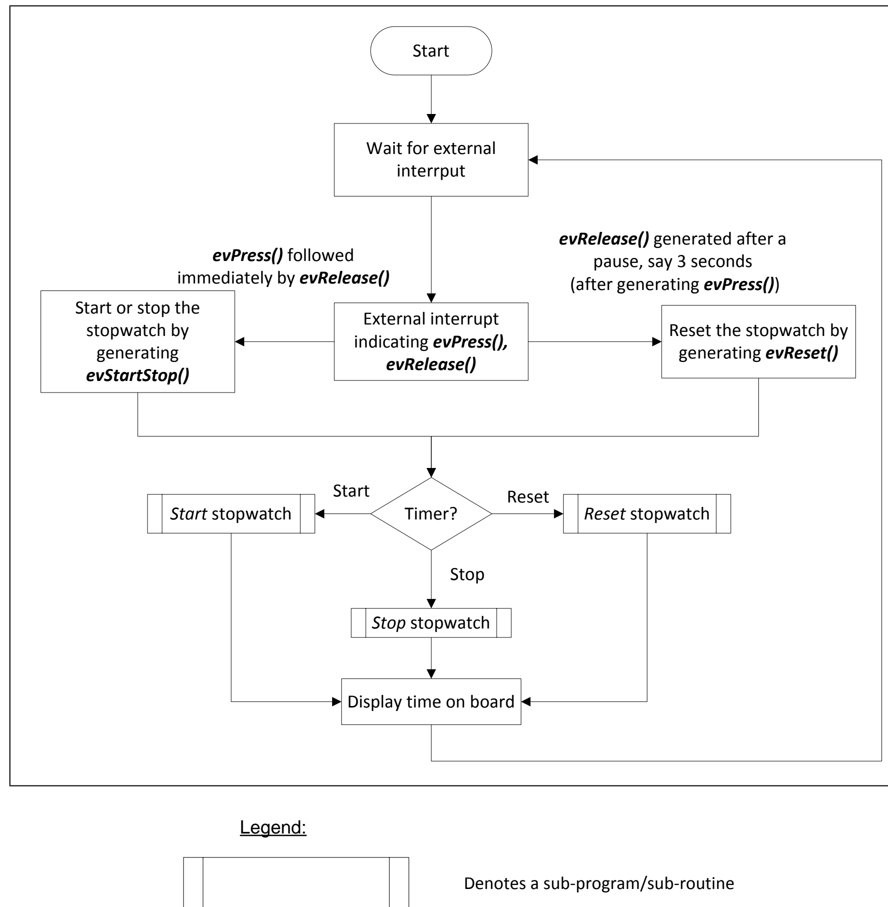



Figure 5.22: Functionality of a stopwatch embedded software application

separate sub-routines. This is highlighted using a rectangular box for sub-routines as seen in Figure 5.22.

The aforementioned functionality of the stopwatch embedded software application are implemented as a prototype using the Model Driven Development (MDD) methodology illustrated in Figure 3.1 (left side). Based on the requirements, the design model of the stopwatch application scenario is specified using UML diagrams in a MDD tool, Rhapsody [42]. The design model of the embedded software for the stopwatch application comprises of two classes, namely *Button* and *Stopwatch*. The detailed functionality of these classes and the reactive behavior is implemented using UML state charts for these classes. In the next step, the system code is automatically generated based on the design model specified in the MDD tool. The system code (stopwatch embedded software application) thus obtained is deployed on the evaluation board. Once the system code is deployed on the embedded device, the application runs in a forever/infinite loop.

The stopwatch embedded software application, thus developed, is subject to quality assurance using the proposed model-based test automation approach. For the test framework generation complexity measurements, example scenarios from the stopwatch application for generating the test framework for a chosen SUT (e.g. *Button*, *Stopwatch* classes) are considered. Note that the examples based on the stopwatch application correspond to scenarios 1 and 2 in Table 5.6.

#### 5.3.1.b Spark extinguishing system example

The spark extinguishing system example is based on a real-life industrial embedded system project. *GreCon* [33] manufactures spark extinguishing systems for (among others) the wood processing industry. Such systems are complex, consisting of many spark detectors (e.g. Figure 5.23) centrally connected to fire extinguishing machine(s).

Figure 5.23: Mounted spark detector [33]

The core of a spark extinguishing system is the control console, where the signals of the individual spark detectors are processed. The control console consists of an embedded system, which in turn hosts a resource-constrained 16-bit microcontroller [22]. The control console performs the processing of the signals received from the spark detectors and initiates the respective tasks to be carried out by the spark extinguishing system. For example, among other functionalities, the control console handles events such as alarm, shutdown and faults. The entire development of this system is certified to safety standards, therefore requiring thorough quality assurance procedures. The development of the embedded software in the control console of the spark extinguishing system is based on the MDD phase described in chapter 3 (left side of Figure 3.1). During the MDD phase, based on the requirements specification the design model for the embedded software to be developed for the control console is specified using an industry standard MDD tool, Rhapsody [42]. Figure 5.24 presents an overall view of the design model in the embedded software developed for the control console.

The data flow is indicated by the arrows labeled as “flow” in Figure 5.24. The individual actors are *Analog Input*, *Digital Output*, *Central Bus*, *Communication Bus* and *Operator*. The individual software modules are grouped in packages based on their respective functionality. The various packages in the design model are *Connector*, *External Interfaces*, *System Master*, *Operations*, *Event Handling* and *Man-Machine Interface (MMI)*. The functionality of the individual actors and packages are described below.

- **Analog Input**: At this level the analog inputs from the spark detectors (e.g. switch, signal generator) are processed and directed to the appropriate interface in the master module.
- **Digital Output**: It receives information about the states to be signaled from the master module. It also signals the received data to the display in the control-board and the LEDs. This indicates the working state of the control console.
- **Central Bus** and **Communication Bus**: The central bus provides communication between the master module and the line modules. The communication bus is responsible for the communication between the master module and the external devices. Both handle a CAN bus [83] using a CANOpen protocol [83].

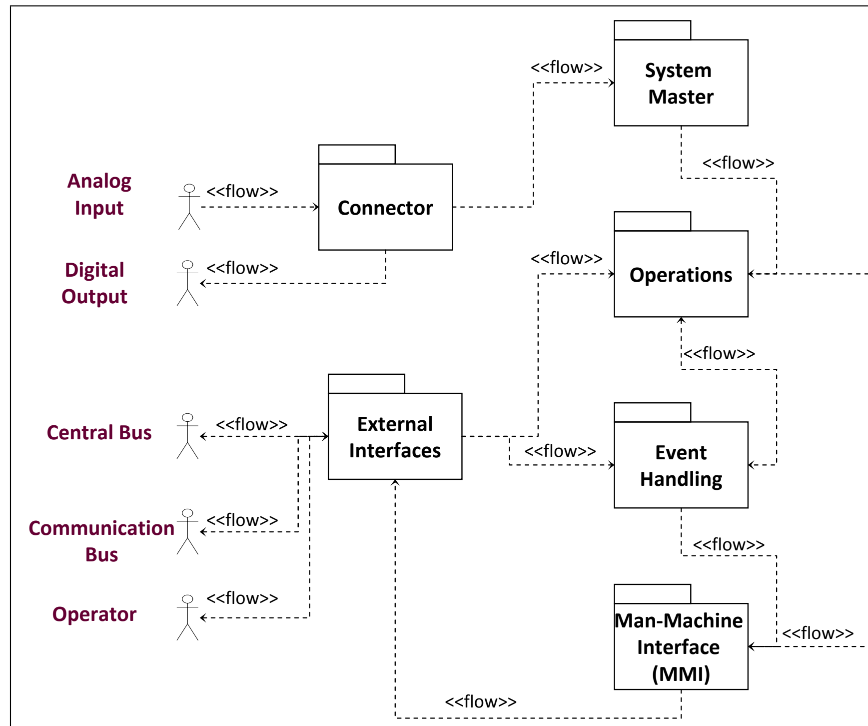



Figure 5.24: Overall view of the design model for the spark extinguishing system example

– **Operator**: The operator makes entries via the control board, for example to handle the LED display in the control console.

– **Connector**: The modules in this package are the interfaces to the Input/Output level. Analog input data from external source and digital output to the control board are managed in this package.

– **External Interfaces**: The CAN network layer of the developed embedded software is handled in this package. All the data exchanged using the CANOpen protocol is realized by appropriate functions.

– **System Master**: This package consists of modules for the initialization of the application and functionality to manage the system functions.

– **Event Handling**: This package is responsible for the management of current events (alarms, faults or shutdown). The respective output is directed to the Man-Machine Interface (MMI) package.

– **MMI**: This provides the interfacing functionality between the operator and the spark extinguishing system. It consists of the display, push button and the LED components of the control board.

– **Operations**: The modules of this package implement the functionality to control the operating level of the system. The system can be in one of the following operating states, namely, *alarm*, *fault*, *shutdown* and *idle*. The processing of user actions (based on external input from control-board/display unit) and the output of the system information is handled in this package. Functionality to display the possible system states (e.g. current operating state) is also included in this package.

For the various functional packages and modules in the spark extinguishing system example, state charts are created to describe their detailed functionality and reactive behavior. Once the design model is complete, the next step is the automatic code generation process based on the design model. The system code thus obtained is executed on the control console, which is the embedded system under evaluation in this scenario. An elaborate description of the spark extinguishing system case study is available in an already published work [47].

For the empirical evaluation of the test framework generation algorithm (Table 5.6), some example scenarios from the *Operations* module (described above) are used as input for the test framework generation algorithm. For instance, the scenarios 5, 6, 7, 8 and 9 indicated in Table 5.6 are based on examples from the *Operations* module in the spark extinguishing system example.

#### 5.3.1.c Experimental setup for determining the *generation* complexity parameters

In order to analyze the test framework generation complexity, the test framework generation algorithm is used to generate the necessary test artifacts (i.e., the test framework) to execute the model-based test cases in the embedded system for various scenarios. The test framework generation algorithm is applied (to generate the test framework) for various scenarios from the aforementioned example case studies such as (a) MIDI system analyzer, (b) stopwatch application and (c) spark extinguishing system.

The test framework generation algorithm is executed on a standard x-86 based computer (with *Intel Core2Quad CPU*, *2.83GHz*, *3.25 GB RAM*) running the Windows XP operating system. In this setup, the time taken to execute an unit operation (involving Java code) is approximately 500 ns (obtained by measurement). Results based on applying the test framework algorithm for generating the test framework for the three example scenarios mentioned above is shown in Table 5.6.

Table 5.6: Empirical results for test framework generation complexity for various scenarios

| Scenario | Elements in SUT (design model) | | | | | Generation complexity | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **a** | **c** | **e** | **s** | **t** | $\beta$ (s) | $\delta$ (s) | $\gamma$ (s) | $\Gamma_g^T(s)$ | $\Gamma_g^S$ (MiByte) |
| 1 | 1 | 2 | 1 | 4 | 5 | 2.88 | 0.41 | 8.27 | 11.56 | 0.60 |
| 2 | 1 | 2 | 2 | 2 | 2 | 3.81 | 0.37 | 12.79 | 16.97 | 0.62 |
| 3 | 1 | 2 | 3 | 8 | 12 | 5.05 | 0.37 | 16.14 | 21.56 | 0.73 |
| 4 | 2 | 3 | 3 | 2 | 2 | 5.11 | 0.36 | 18.14 | 23.61 | 0.72 |
| 5 | 3 | 4 | 7 | 11 | 21 | 46.57 | 0.60 | 389.37 | 436.54 | 0.72 |
| 6 | 3 | 4 | 9 | 14 | 28 | 45.67 | 0.60 | 414.40 | 460.67 | 0.75 |
| 7 | 4 | 4 | 13 | 16 | 36 | 46.31 | 0.67 | 465.44 | 512.42 | 0.76 |
| 8 | 6 | 7 | 12 | 45 | 124 | 57.75 | 1.03 | 470.01 | 528.79 | 0.96 |
| 9 | 12 | 13 | 33 | 29 | 106 | 78.02 | 1.36 | 715.96 | 795.34 | 1.21 |

a-Number of association ends, c-number of associated classes,
s, $t$ and $e$: Number of states, transitions and events in state chart of the SUT.

$\beta, \gamma, \delta, \Gamma_g^T, \Gamma_g^S$ are average values from three runs of the algorithm for each scenario.

The example scenarios consist of a varying number of association ends ($a$) and classes ($c$) participating in association ends for a chosen SUT. Similarly, the number of events ($e$), states ($s$) and transitions ($t$) for a given SUT are different for each scenario as shown in the Table 5.6.

As mentioned earlier, scenarios 1 and 2 correspond to examples from the stopwatch embedded software application (section 5.3.1.a). Scenarios 3 and 4 correspond to examples from the MIDI system analyzer case study (section 4.1). Similarly, scenarios 5, 6, 7, 8 and 9 correspond to examples from the spark extinguishing system case study. Thus, the scenarios 1-9 in the empirical evaluation present a subset of varying complexity (i.e., varying number of associated classes, states and events) for the SUT under consideration. Therefore, the selected scenarios provide a reasonable subset of real-life examples for experimental evaluation of the test framework generation algorithm and determination of the test framework generation complexity (time and memory-size).

#### 5.3.1.d Generation time and memory-size complexity analysis

To understand and analyze the empirical results from the test framework generation-complexity measures, the theoretical complexity analysis provided in section 4.5 can be recalled. Based on the theoretical analysis of the complexity measures, an expression for the time and memory complexity of the test framework generation algorithm are stated as $\boxed{\Gamma_g^T=\beta+\gamma+\delta}$ and $\boxed{\Gamma_g^S=\beta+\gamma+\delta}$ respectively. In the above expressions, $\beta$, $\gamma$ and $\delta$ denote the respective processing complexities for the generation of the three components of the test framework namely, the *proxy test model*, *communication interface* and the *UTP artifact* respectively. Similarly, $\Gamma_g^T$ and $\Gamma_g^S$ denote the approximate time and memory *generation* complexity of the test framework generation algorithm (algorithm 1 in section 4.4.1).

The approximate theoretical estimates of the generation complexity for the three components of the test framework can be deduced from the theoretical estimates in section 4.5. The approximate generation-time complexity measures in $\Gamma_g^S=\beta+\gamma+\delta$ are $\beta = n^2$, $\gamma = n^3$, $\delta = n$. In other words, by the theoretical estimates of the complexity measures, a major portion of the total time for the test framework generation ($\Gamma_g^T$) is expected to be consumed by the time required for generating the communication interface component ($\gamma$), followed by proxy test model component generation time ($\beta$) and finally the time required to generate the UTP artifact component ($\delta$) of the test framework. These approximate generation-time complexity measures are recalled here to provide a comparative study with the empirical results available in Table 5.6.

For the various scenarios shown in Table 5.6, $a$ and $c$ denote the number of association ends and associated classes respectively, for the chosen SUT (in a given scenario). Similarly, $s$, $t$ and $e$ denote the number of states, transitions and events in the state chart of the chosen SUT. For the aforementioned scenarios (1-9), Table 5.6 shows the approximate generation time complexity measures, namely, the individual components $\beta$, $\gamma$, $\delta$ (in seconds), the total time required $\Gamma_g^T$ (in seconds) and the memory complexity measure $\Gamma_g^S$ (in Megabytes). These values ($\beta$, $\gamma$, $\delta$, $\Gamma_g^T$ and $\Gamma_g^S$) are the average of three runs of the test framework generation algorithm, obtained by measurement on the Windows-based host. The results correspond to the test framework generation complexity for one SUT.

Based on the experimental results available in Table 5.6, the following conclusions can be made. For the generation time complexity estimates ($\Gamma_g^T$), approximately 70% to 90% of the total test framework generation time is spent for generating the *communication interface* component of the test framework ($\gamma$), as expected. The procedure in algorithm 4 thus consumes a major chunk of the total time for generating the test framework on the host computer. This can be attributed to the generation of

the decoding functionality in the communication interface component ($T_{decode}^{Com}$) of the test framework (algorithm 4). The functionality of the communication interface component and its decoding features (i.e., support for decoding the test result from the target) are discussed in detail in section 4.4.5. Based on the code-snippet available in section 4.4.5, it is evident that in order to decode the incoming test result, string compare statements between the event name and the class name (one time each for source and destination) are required. Therefore, a major part of the test framework generation time is spent in order to generate these decoding statements for the *decodeMsgFromTrgtDebg()* ($T_{decode}^{Com}$) operation in the communication interface component. This empirical result is in line with the approximate time complexity predicted (theoretically) for the time involved in generating the communication interface component ($\gamma \approx n^3$).

As seen in Table 5.6 the next major component in the total time taken by the test framework algorithm, to generate the test framework ($\Gamma_g^T$), is the *proxy test model* component ($\beta$). The proxy test model creation time includes the time to generate the proxy classes (lines: 1-9 in algorithm 1) and the state chart of the SUT in the proxy test model (algorithm 2). It also includes the time for generating the *generateEventConsumed()* operation for each proxy class. This function is used by the proxy classes to invoke the respective operations (among the proxy classes), in order to mirror the test case execution process at the embedded system. However, the generation time required for generating the proxy test model component of the test framework ($\beta$) does not grow as significantly as that of the communication interface component ($\gamma$). These conclusions are in accordance with the theoretical estimates for the generation time complexity of the communication interface component ($\gamma \approx n^3$) and that of the proxy test model component ($\beta \approx n^2$) in the test framework.

The time required to generate the UTP artifact component ($\delta$) is the least among the time taken to generate the components of the test framework (i.e., algorithm 3). From the results in Table 5.6, it is also evident that the processing requirements and the time spent for generating each component grows with the increasing number of input elements such as association ends, associated classes and events for the chosen SUT. These results are in accordance with the inferences from the theoretical estimates for the test framework generation time complexity, briefly outlined above (and discussed in detail in section 4.5).

The measure of the memory required by the test framework generation algorithm (algorithm 1) to generate the test framework, on the host computer, is denoted by $\Gamma_g^S$ in MiByte[1]. From the values in Table 5.6, it is clear that the approximate memory requirement for the test framework generation algorithm is bounded on the host computer. For the various scenarios in Table 5.6, the approximate memory consumption during the test framework generation is between 0.6 and and 1.21 MiByte.

In summary, the time and memory complexity measures for the various scenarios indicate that the time and memory required on the host for test framework generation (corresponding to a chosen SUT and design model) increases with the increase in the number of associated classes and events for a chosen SUT. These observations from the experimental analysis are complementary to the complexity estimate derived theoretically in equations 4.3 and 4.4 (in section 4.5-chapter 4).

[1] 1 MiByte=1024 KiBytes

### 5.3.2 Runtime complexity

An empirical analysis based on the experimental results for the time and memory complexity measures (during runtime) is provided in this section (challenge (3)). A theoretical analysis on the time and memory complexity measures while executing the model-based test cases in the embedded systems (i.e., during runtime) using the proposed test framework approach is provided in section 4.5.

This section is organized as follows. An empirical analysis of the runtime-time complexity is provided in section 5.3.2.a. Similarly, an analysis of the runtime-memory complexity is provided in section 5.3.2.b. In order to analyze the empirical results for the runtime complexity, the complexity measures derived using a theoretical analysis (in section 4.5) are recalled. For determining the various parameters during runtime complexity, examples from the MIDI system analyzer case study are used (e.g test cases, *MonitorIO* routine for EIA-232 debug interface, etc)

#### 5.3.2.a Time complexity

Let us recall the expression for runtime-time complexity measure derived using a theoretical analysis in section 4.5. The time complexity measure ($\Gamma_r^T$) during runtime for executing the model-based test cases using the proposed approach can be stated as,

$$\Gamma_r^T = n_{tc} \cdot n_{dr} \cdot A_{sut} \cdot [\overbrace{Tcp_{send} + Deb_{send} + Deb_{recv} + Tcp_{recv}}^{\text{host computer}}$$
$$+ \overbrace{E_{decode} + E_{exec} + E_{mon}}^{\text{embedded system}}]$$

As seen above, the time complexity ($\Gamma_r^T$) during runtime comprises of two main parts, namely the time consumed on the host computer and the time consumed on the embedded system. However, the total time spent (on the host and the embedded system) for executing the model-based test cases using the proposed approach, depends on the number of test cases $n_{tc}$ to be executed. Each test case with $n_{dr}$ test drivers involves invoking $A_{sut}$ actions.

The time spent on the host computer for executing the test cases is a sum of the time spent to send and receive the test data (i.e., test input data and test results) for each test case between the host and the embedded system. The test data is communicated by the TCP/IP communication interface in $Tcp_{send}$ time units and the target debugger in $Deb_{send}$ time units, to the embedded system. Similarly, the test case result is sent and interpreted in $Deb_{recv}$ and $Tcp_{rec}$ time units respectively on the host computer.

The main goal of this thesis is to propose a test automation approach, wherein there is only minimal and bounded overhead on the embedded system for executing the model-based test cases. Therefore, in the proposed test framework approach, the test cases are executed on the host computer and only the test input data is executed on the embedded system (challenges (2) and (3)). Then, the time spent in the embedded system during the execution of the test cases (at runtime) only comprises of the time to decode the test input data ($E_{decode}$), execute the test input data ($E_{exec}$) and send the test result (e.g. event consumed notification) using the target monitor routine ($E_{mon}$). To gain further understanding on the aforementioned parameters in the time complexity measure ($\Gamma_r^T$), the following provides an insight on the measurement and analysis of these parameters for example scenarios.

### Determining $Tcp_{send}$ and $Tcp_{recv}$ on the host computer

The approximate time taken for sending and receiving messages between the test framework and the target debugger, namely $Tcp_{send}$ and $Tcp_{recv}$, using the TCP/IP communication interface is measured on the host computer. The test framework is generated for the example application scenarios on a x-86 based PC (with *Intel Core2Quad CPU*, *2.83GHz*, *3.25 GB RAM*), running Windows XP operating system. The timer granularity of this system is set to 15.6 ms by default. Therefore, to obtain a finer granularity of 1 ms (i.e., for determining $Tcp_{send}$ and $Tcp_{recv}$), functions such as *QueryPerformanceCounter()* (returns the number of *ticks* since the computer was rebooted) and *QueryPerformanceFrequency()* (returns the number of *ticks* in a second) [74], [82] for the Windows operating system are employed. Note that this method can be considered as one among the several alternatives to obtain a finer timer granularity on a Windows-based host.

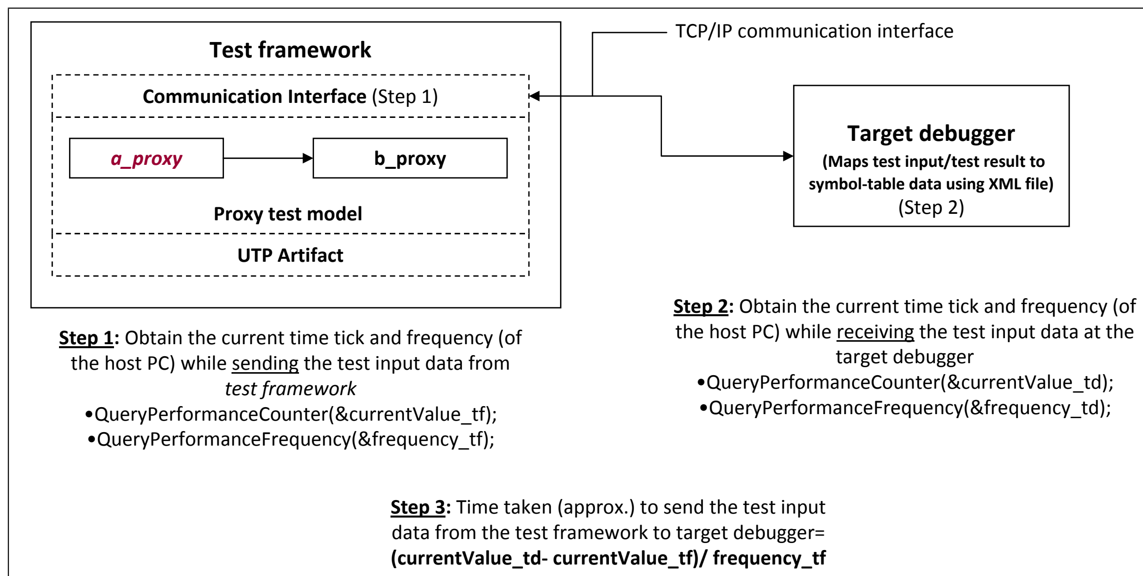



Figure 5.25: Steps for determining the time required for sending the test input data between the test framework and the target debugger ( $Tcp_{send}$)

Therefore the approximate time values on the Windows host are obtained by the following measurement. For example, to measure the time taken to send the test input data from the test framework to the target debugger (i.e., $Tcp_{send}$), the following technique is used. The current time tick when the test input data is sent from the test framework (*QueryPerformanceCounter(currentValue_tf)*) and the current time tick at which the test input data is received at the target debugger *QueryPerformanceCounter(currentValue_td)* are obtained by measurement. Similarly, the number of *ticks* in a second configured for the Windows-based host on which the measurements are made is obtained (*QueryPerformanceFrequency(frequency_tf)*). These steps are shown as steps 1 and 2 in Figure 5.25. By dividing the difference in the time between between sending and receiving the test input data (i.e., elapsed time) by the frequency of the time ticks (obtained using *QueryPerformanceFrequency(frequency_tf)*), the approximate time to send/receive the test data between the test framework and the target debugger on the host can be obtained (step 3 in Figure 5.25). For example, the following code (C language) on the host, returns the attributes $currentValue_tf.QuadPart$ and $frequency_tf.QuadPart$ at the *test framework*, when the test data is sent to the target debugger.

```
1 LARGE_INTEGER currentValue_tf, frequency_tf; /* Declare the required variables */
2 QueryPerformanceCounter(&currentValue_tf); /* Obtain the current value */
3 QueryPerformanceFrequency(&frequency_tf);
4 return (currentValue_tf.QuadPart, frequency_tf.QuadPart);
```

Similarly, the following code returns the attributes $currentValue_td.QuadPart$ and $frequency_td.QuadPart$ at the target debugger, when the test data is received at the target debugger. Note that the values $frequency_td.QuadPart$ and $frequency_tf.QuadPart$ are obtained to verify if they return the same value.

```
1 LARGE_INTEGER currentValue_td, frequency_td; /* Declare the required variables */
2 QueryPerformanceCounter(&currentValue_td); /* Obtain the current value */
3 QueryPerformanceFrequency(&frequency_td);
4 return (currentValue_td.QuadPart, frequency_td.QuadPart);
```

Once these values are obtained by measurement, the following formula gives the time elapsed between the sending of the test data at the test framework and the reception of the test data at the target debugger.

$$Tcp_{send} = \frac{currentValue_td.QuadPart - currentValue_tf.QuadPart}{frequency_td.QuadPart}$$

As stated earlier, *frequency_td.QuadPart* or *frequency_td.QuadPart* can be used in the above formula as they are expected to be the same (as the measurement is made on the same host). Similar steps are used to determine the value of $Tcp_{recv}$. However, both $Tcp_{send}$ and $Tcp_{recv}$ are approximate values and vary over several iterations as the test framework runs on a Windows-based host. Empirical values for these parameters while executing the test case specified in the example shown in Figure 5.12 are listed in Table 5.7. These measurements correspond to the test input data (corresponding to five events injected as test input) from the test driver to the SUT, in the test case shown in Figure 5.12.

Table 5.7: Parameters in time-complexity $\Gamma_r^T$ (runtime) for the test case specified in Figure 5.12

| **Host computer** | | | | **Embedded system** | | |
|---|---|---|---|---|---|---|
| $Tcp_{send}$ (ms) | $Tcp_{recv}$ (ms) | $Deb_{send}$ (ms) | $Deb_{recv}$ (ms) | $E_{decode}(\mu s)$ | $E_{exec}(\mu s)$ | $E_{mon}(\mu s)$ |
| 1.4 | 2.2 | 1.5 | 2 | 5 | 9.4 | 74.52 |
| 4.8 | 3.4 | 1.5 | 2 | 5 | 9.4 | 74.52 |
| 4.7 | 4.5 | 1.5 | 2 | 5 | 9.4 | 74.52 |
| 1.6 | 1.8 | 1.5 | 2 | 5 | 9.4 | 74.52 |
| 3.8 | 4.3 | 1.5 | 2 | 5 | 9.4 | 74.52 |

#### Estimation of $Deb_{send}$ and $Deb_{recv}$ on host computer

As mentioned before, during the runtime complexity measurements various examples from the MIDI system analyzer case study are considered for experimental evaluation. For the MIDI system analyzer example, the serial/EIA-232 interface is chosen as the communication interface between the host and

the target. Hence, in this empirical analysis the values for $Deb_{send}$ and $Deb_{recv}$ are determined based on the EIA-232 interface.

$Deb_{send}$ denotes the time taken by the target debugger to send the test input data to the embedded system via the debug interface. Similarly, $Deb_{recv}$ represents the time to send the test results (e.g. event execution results via event consumed notification) to the target debugger, using a debug communication interface. These time values can be calculated based on the speed of the data transfer using the debug interface and the number of bytes required to be transferred.

For the EIA-232 interface used in the MIDI system analyzer example, a baud rate of 115200 is used (as this is an often used baud rate for the EIA-232 interface). Consider an example of test input data sent from the target debugger to the embedded system in the MIDI system analyzer example, in the test case shown in Figure 5.12. It can be recalled that for sending the test input data corresponding to injecting an event *evStatusMsg(MIDI_Cmd=144)* from the test environment (*TestEnv*) to an instance of the *PreDecoder* class namely *itsPreDecoder1* on the target, the test input data is represented as *<evStatusMsg TestEnv itsPreDecoder1 144>*. Then the total frame length for sending this input data is 18 bytes (Figure 5.7, Table 5.2). Therefore, the target debugger needs to send 18 bytes of test input data for the above example. This test input data is sent to the embedded system using the debug communication interface (i.e., EIA-232). Similarly, the test result (i.e., event consumed notification) comprises of 23 bytes of data sent to the target debugger by the target monitor, for this example (*evStatusMsg TestEnv itsPreDecoder1 230 CMD_Received MIDI_Cmd=144*). Hence, based on the number of bytes to send and receive the test input data, at a chosen data rate (115200 baud), the values for $Deb_{send}$ and $Deb_{recv}$ are calculated as follows.

$Deb_{send}$ and $Deb_{recv}$ are calculated as the inverse of the baud rate divided by the number of bytes to send over the interface. Thus, the inverse of $[115200/(1 + 8 + 1)/18]$ and $[115200/(1 + 8 + 1)/23]$ provides the time taken to send 18 and 23 bytes over the EIA-232 interface, with 115200 baud, which has 1 start bit, 8 data bits and 1 stop bit for the data transfer. Thus, the time to send the test input data (18 bytes) and the time to receive the test results (23 bytes) over the EIA-232 interface in the MIDI system analyzer are estimated as $Deb_{send} = 1.5ms$ and $Deb_{recv} = 2ms$ (Table 5.7).

### Determination of $E_{decode}$, $E_{exec}$ and $E_{mon}$ in the embedded system

By definition in the complexity measures, $E_{decode}$ denotes the time taken by the target monitor to decode the incoming test data (sent by the target debugger). Similarly, $E_{exec}$ denotes the generation and consumption time for the decoded event in the RTOS framework used and $E_{mon}$ is the time spent in the monitor routine to send the event consumed notification (i.e., in this case the test result) to the target debugger. These values are measured using an oscilloscope [101] in the embedded system. Note that the target executes the application software (MIDI system analyzer) bundled with the target monitor routine (for EIA-232 interface) and the RTOS used in the prototype (namely OORTX-RXF).

By measurement (using an oscilloscope [101]), these values are determined as $E_{decode} = 5\mu s$, $E_{exec} = 9.4\mu s$ and $E_{mon} = 74.52\mu s$ (for EIA-232 interface) as seen in Table 5.7. Note that the values $E_{decode}$ and $E_{exec}$ are independent of the debug interface used and the application running on the target. But, they depend on the RTOS framework under consideration (in the prototype the OORTX-RXF). However, for a chosen RTOS these values are expected to be static, as the underlying operating system is a deterministic real-time operating system.

**Analysis of the parameters in runtime-time complexity ($\Gamma_r^T$)**

For a comparative study, consider the various parameters in the runtime complexity measure $\Gamma_r^T$ obtained by measurement during runtime, involving the test case shown in Figure 5.12. The empirical values obtained by measurement for the various parameters in $\Gamma_r^T$ are shown in Table 5.7. Based on the values of the parameters in Table 5.7, it is clear that the varying parameters during runtime involving the execution of the model-based test cases using the automatically generated test framework are $Tcp_{send}$ and $Tcp_{recv}$. The time taken to send/receive the test input data between two processes (i.e., the test framework and the target debugger) varies based on the host computer (e.g. load), running a Windows-based OS. This is also evident from the empirical values for $Tcp_send$ and $Tcp_recv$ in Table 5.7. However, this does not affect the test case execution process, while executing the functional test cases. While executing performance-based test cases such as the ones involving the verification of timing constraints over several events, the parameter $Tcp_{send}$ could influence the test case execution process. This is because of the delays and variation in delays (jitter) introduced by $Tcp_{send}$ while sending the test data to the target debugger.

On the other hand, the parameter $Tcp_{recv}$ gains no significance in either functional testing or while verifying timing constraints at the target using the test framework. This is because, the time taken to receive the test result at the test framework from the target debugger has no impact in determining the test case execution results. As the test case execution results are already available at the test framework the time to receive the test results at the test framework $Tcp_{recv}$ gains no significance (in $\Gamma_r^T$) during the test case execution process.

Based on the empirical analysis it is clear that except for $Tcp_{send}$, the parameters in $\Gamma_r^T$ are all static, bounded and are measurable beforehand. Therefore, the parameters $Deb_{send}$, $Deb_{recv}$, $E_{decode}$, $E_{exec}$ and $E_{mon}$ are all constant/static (i.e., do not vary during runtime) and measurable beforehand. These values can be included in the test cases, while specifying non-functional tests such as the test cases to verify the time-related aspects of the embedded software. The time delay introduced by $Tcp_{send}$, i.e., time taken to send individual messages with test data to the target debugger can also be ignored while verifying the non-functional test cases in the MIDI system analyzer case study. This is because, $Tcp_{send}$ gains significance only during the verification of temporal constraints over several events. Since, this is not the case in the MIDI system analyzer example, the time delay between sending of individual messages with test input data to target debugger from test framework can be ignored.

In summary, based on the runtime-time complexity analysis it can be stated that the parameters in the runtime-time complexity ($\Gamma_r^T$) are measurable and known beforehand. The parameters on the host computer, namely $Tcp_{send}$ and $Tcp_{recv}$ vary on the host because of the nature of the operating system running on the host computer. On the other hand, the complexity parameters on the embedded system namely, $E_{decode}$, $E_{exec}$ and $E_{mon}$ are static, bounded and deterministic as they are measured on a deterministic real-time operating system (e.g. OORTX-RXF is the RTOS used in the prototype). Moreover, since the time delay introduced in the embedded system using the proposed approach is in the order of $\mu s$, there arises an opportunity for the end user to include this delay in the earlier phases of the development cycle. By this, a minimally intrusive runtime monitoring mechanism, which is time-aware can be applied for executing the model-based test cases in the embedded system. This is a significant improvement over the existing approaches which introduce unbounded overhead parameters

during the test case execution process at the embedded system (challenge (3)).

An example pertaining to the verification of non functional requirements, such as temporal properties, using the test results available at the target debugger GUI (in the proposed approach) is discussed below (section 5.3.2.b).

#### 5.3.2.b Verification of non-functional requirements - An example

Verification of functional requirements using the proposed approach has been discussed in section 5.2.3.d. Aspects related to non-functional testing such as the verification of temporal requirements can be carried out using the results received from the target monitor. Based on the application of the target debugger GUI, discussed in section 5.1.1.b (Figure 5.4, Figure 5.5, Figure 5.6), it is clear that the target debugger can also be used to visualize the target behavior in real time using UML sequence diagrams (with time stamps) and timing diagrams.

On the other hand, whereas the test framework is used to display the test results using UML sequence diagrams, the temporal constraints have to be verified manually at the target debugger GUI. This is because, at this juncture the prototype implementation (for generating the test framework automatically) does not include an automatic verification (and displaying at the model level) of the timing properties based on the received results from the target monitor via the target debugger.

An example pertaining to the visualization of the test case execution result in the target debugger GUI (corresponds to the test case/test input data in Figure 5.12) and visualizing the same scenario by pressing a key on the key board (envisaged functionality) is shown in Figure 5.26-(a) and (b) respectively. From Figure 5.26-(a) and (b), it is clear that the test input data (i.e., during the test case execution on the target) is injected from the external test environment as indicated by the lifeline of the object *TestEnv*. On the other hand, when visualizing the target behavior in real time, the test input messages for the instance of the *PreDecoder* class namely, *itsPreDecoder1* are injected from the target (i.e., based on the processing of the input messages from the MIDI keyboard by the MIDI system analyzer application).

#### Temporal requirements verification

Consider the verification of non-functional aspects for the MIDI system analyzer using the model-based results available at the target debugger GUI (Figure 5.26-(a)). The non-functional aspects for the MIDI system analyzer pertains to the following temporal requirements, namely: (a) the maximum latency/delay from a keystroke to a sound has to be less than 5 ms and (b) the maximum jitter (difference in delay) between the keystrokes and the sound generation has to be less than 2 ms.

From the results of the test case execution visualized in the target debugger GUI, both these temporal (non-functional) requirements can be verified manually. Note that while executing the above test case using the test framework, the timer granularity of the RTOS used in the prototype, i.e., OORTX-RXF, is configured to 0.1 ms (one system tick in the RTOS corresponds to 0.1 ms).

Consider the test case specified in Figure 5.12. The real time requirement (a) which states that the time elapsed between the reception of a key press message to the sound generation should be less than 5 ms, can be mapped to this test case as follows. The time elapsed between the injection of the event *evDataByte* with parameter *Byte=127* (from *TCon_PreDecoder_proxy.itsTestDriver_of_PreDecoder_proxy* to *TCon_PreDecoder_proxy.itsPreDecoder_proxy*) and the generation of

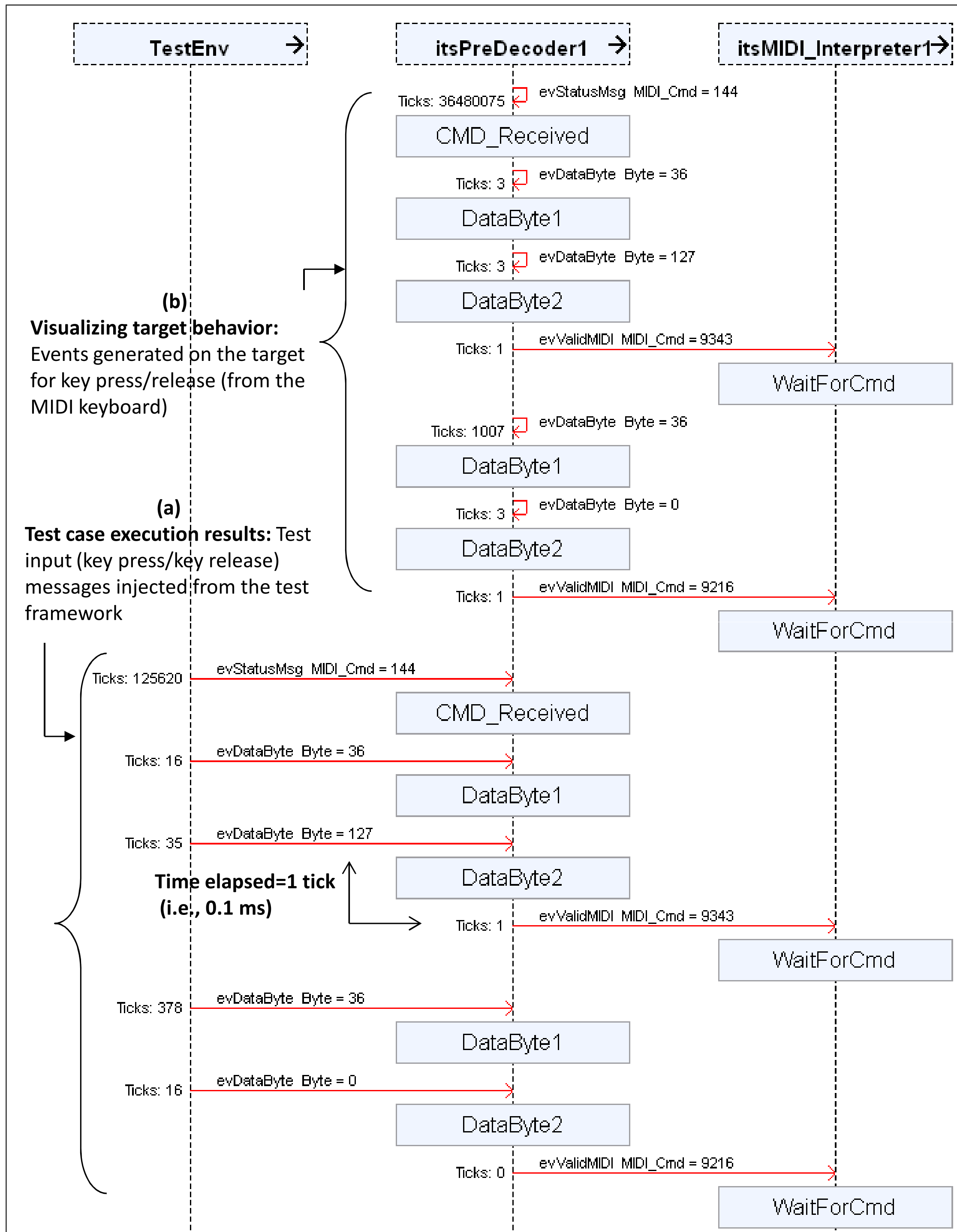



Figure 5.26: (a) Test case execution result in the target debugger GUI (corresponds to the test case/ test input data in Figure 5.12) and (b) visualizing the same scenario by pressing a key on the key board (envisaged functionality), for the MIDI system analyzer

the event *evValidMIDICmd* with parameter *MIDI_Cmd=9343* (from *TCon_PreDecoder_proxy.itsPreDecoder_proxy* to *TCon_PreDecoder_proxy.itsMIDI_Interpreter1*) is expected to be less than 5 ms.

The results of the above test case execution (using the proposed approach) are visualized using the target debugger GUI as shown in Figure 5.26-(a). From Figure 5.26-(a), it is clear that the time elapsed between the injection of the event *evDataByte* with parameter *Byte=127* (from *TestEnv* to *itsPreDecoder1*) and the generation of the event *evValidMIDICmd* with parameter *MIDI_Cmd=9343* (from *itsPreDecoder1* to *itsMIDI_Interpreter1*) is 1 tick (which corresponds to 0.1 ms, because of the configured timer granularity of the RTOS). This value is less than the temporal requirement (a) specified for the test case, namely 5 ms.

Similarly, based on the execution of the test case in example 2 (Figure 5.15), i.e., key press/release messages sent over five iterations, the variation in delay between several keystrokes and sound generated can be verified. For the test case example 2 in Figure 5.15, the delay between the keystroke messages for five iterations is obtained as 0 ms. Hence, the variation in delay between these keystroke messages can be verified manually as less than 2 ms (verifies performance requirement (b)).

#### Comparison with existing techniques in MBT tools

Based on the above discussion, it is clear that the non-functional properties such as the timing requirements of the MIDI system analyzer can be verified manually based on the test execution results available in the target debugger GUI. The time spent in the monitor routine is bounded, measurable and known before hand. For the given test case, the temporal requirement involves a timing constraint of 5 ms. The test result visualized in the target debugger shows that the requirement is guaranteed as the time elapsed for the given requirement is 0.1 ms. Note that this time value includes the total time spent in the embedded system for executing a test data corresponding to the test input data (i.e., 88.92 $\mu s$). Therefore, even the timing properties of the embedded software can be verified using the time-aware runtime monitoring approach used in the proposed test framework approach. Thus, in the case of test cases involving verification of temporal requirements over a series of events, the static/bounded delays introduced by the proposed approach can be included in the test case. This is a significant advantage in comparison with the existing approaches in MBT tools such as [43] and [44].

In the existing approaches, the test harness is downloaded on the target for the purpose of test case execution and removed once the test case execution process is complete. Moreover, the delays introduced by the test harness code is neither bounded nor measurable beforehand. Since the test harness required to be downloaded in the target is significant, the existing approaches are not applicable for resource constrained embedded systems. This implies that the existing approaches are not only limited in terms of their applicability but also may result in an erroneous test case execution process while verifying the real time properties of the embedded software.

#### 5.3.2.c Runtime-memory complexity

Let us recall the runtime-memory complexity measure ($\Gamma_r^S$), obtained using a theoretical analysis in section 4.5. This complexity measure can be stated as

$$\Gamma_r^S = \overbrace{\overbrace{[A_{sut} \cdot S_{TF}] + [n_{tc} \cdot n_{dr}]}^{\text{test framework}} + \overbrace{S_{TD}}^{\text{target debugger}}}^{\text{host computer}} + \overbrace{S_{TM}}^{\text{embedded system}}$$

It is intuitive to perceive that the runtime-memory complexity measure comprises of two main components, namely, the memory requirement on the host and the memory requirement on the embedded system while executing the model-based test cases using the proposed approach. This is reflected in the theoretical estimate of the runtime-memory complexity measure ($\Gamma_r^S$) shown above.

The memory requirement on the host computer during runtime for the proposed approach comprises of the memory requirement for the test framework (test harness generated for the automatically generated test framework and manually specified test cases in the test framework) and the memory requirement for the target debugger. One of the main goals of this thesis is to propose a model-based test automation approach with predictable and bounded runtime-time and memory overhead parameters. This is especially envisaged for the overhead parameters in the embedded system (challenge (3)). In this context, the proposed test automation approach is expected to result in bounded and measurable runtime-time and memory overhead parameters. In addition the overhead introduced in the embedded system during runtime (time and memory) is expected to be not only static/bounded and measurable beforehand but also minimal in nature.

Based on the theoretical complexity analysis on the runtime-memory size complexity, the memory requirement for the proposed approach on the host computer is represented as $[A_{sut} \cdot S_{TF}] + [n_{tc} \cdot n_{dr}]$. This comprises of the storage requirement for the automatically generated test framework ($S_{TF}$) for a chosen SUT ($A_{sut}$). Whereas the test framework generated for a chosen SUT has a fixed memory requirement ($S_{TF}$), the memory requirement of the test harness increases in size with the increase in the number of test cases $n_{tc}$ (with test drivers $n_{dr}$) on the host computer.

The target debugger is expected to have a fixed memory requirement on the host computer. This is represented as $S_{TD}$. Thus, the memory requirement on the host computer (during runtime) comprises of the test harness generated (for the test framework and the test cases) and the memory requirement of the target debugger.

On the other hand, the only memory requirement on the embedded system for executing the test input data and sending the test case execution results on the embedded system, using the proposed approach, is that of the memory requirement of the target monitor routine ($S_{TM}$).

Given this background on the theoretical complexity estimates, let us analyze the empirical measurements obtained for the various parameters in the runtime-memory complexity ($\Gamma_r^S$) in detail to gain further understanding.

**Analysis of the runtime - memory complexity parameters on the host ($A_{sut} \cdot S_{TF}$, $n_{tc} \cdot n_{dr}$ and $S_{TD}$)**

In the runtime-memory complexity measure ($\Gamma_r^S$), the component $A_{sut} \cdot S_{TF}$ denotes the memory requirement of the test framework generated for a chosen SUT before the specification of the test cases. This component is expected to have a fixed memory requirement. This is because the test framework with the necessary artifacts, for executing the model-based test cases, is already generated automatically by the test framework generation algorithm ($A_{sut} \cdot S_{TF}$).

The model-based test cases are then specified manually by the end-user in this automatically generated test framework (section 5.2.2). Then the test framework is updated with the test harness required for executing the given number of test cases in the test framework. Note that once the test cases are specified manually in the automatically generated test framework, in the MBT tool [43] in the prototype, the test harness is automatically updated. The test harness required for the test cases

is represented as $n_{tc} \cdot n_{dr}$. In the MIDI system analyzer example (with the chosen SUT (*PreDecoder* class), the test harness generated for test framework (comprising of the static memory requirement of the test framework and the test harness generated for the test cases) is shown in Figure 5.27.

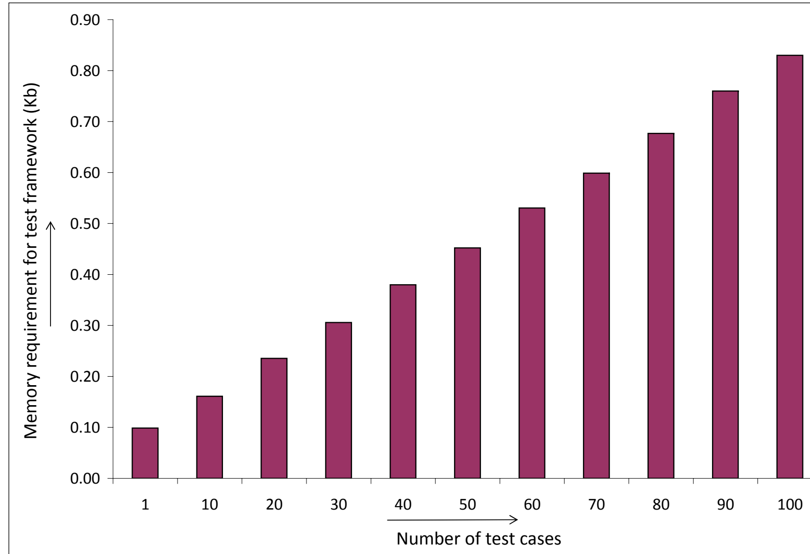



Figure 5.27: Memory requirement for the test framework $[A_{sut} \cdot S_{TF}] + [n_{tc} \cdot n_{dr}]$ on the host computer with increasing number of test cases $n_{tc}$

The memory requirement of the test harness generated for the test framework obtained by measurement is plotted on the vertical axis corresponding to the number of test cases (between 1 and 100) specified along the horizontal axis in Figure 5.27. The test harness size shown in the vertical axis comprises of a static and a dynamic part. The static part comprises of the memory requirement of the test framework before the specification of the test cases. For the chosen SUT (*PreDecoder* class) in the MIDI system analyzer example this value is approximately 100 KiByte (obtained by measurement). This is a fixed memory requirement of the automatically generated test framework. The dynamic part corresponds to the test harness that is updated in the MBT tool after manually specifying the test cases in the automatically generated test framework.

Therefore, in Figure 5.27 the memory requirement for the test framework on the host computer for executing one test case is approximately 100 KiByte. However, the test harness increases with the increase in the number of test cases ($n_{tc}$) specified manually in the automatically generated test framework. On the other hand, an exponential or a significant increase in the test harness code is not observed. This is because, the test harness generated each time for the newly specified test cases in the test framework, comprises of the functionality to include the test drivers (to inject the test input data) for each test case. The test harness generated for the test cases does not require additional test input data based on the test cases specified. It can be recalled that the test input data is already available (i.e., automatically generated) in the state chart of the SUT and it is not duplicated in the test harness when new sets of test cases are specified in the test framework. Thus, the memory requirement on the host computer for the test framework during runtime is not only significantly reduced, but also does not show a significant increase in the test harness code with an increase in the number of test cases.

The target debugger on the host computer has a fixed memory requirement ($S_{TD}$) corresponding

to its executable or DLL file (e.g. in a Windows-based OS) at the host computer. In the prototype implementation of the proposed approach, the memory requirement of the target debugger is 13 MiByte (obtained by measurement).

**Analysis of the runtime - memory complexity parameters on the embedded system ($S_{TM}$)**

The memory requirement in the embedded system ($S_{TM}$) for executing the test routines is among the most important criteria in determining the choice of the automation approach especially for an embedded system (challenge (3)). In the proposed approach, the test framework (automatically generated on the host computer) enables model-based test case execution on the host computer. Whereas, only the test input data corresponding to the test cases are inserted to the embedded system and executed on the embedded system. Then, the incoming test data is injected in the target by the target monitor. The software-based runtime target monitor sends the test case execution results (e.g. event consumed notification) to the host computer via the (chosen) debug interface. Hence, the memory requirement on the embedded system for executing the test cases on the embedded system using the proposed approach is only the memory requirement of the target monitor routine, $S_{TM}$.

The memory requirement for the target monitor routine for the three debug interfaces used in the prototype evaluation is shown in Table 5.4. From this it is clear that the size of the target monitor is static and remains independent of number of the test cases and their complexity. This is primarily because, the test harness is not downloaded on the target for executing the test cases in the embedded system. Instead the test input data alone is executed on the embedded system with the aid of the runtime monitor routine in the target.

For the MIDI system analyzer example, the EIA-232 interface is used as the debug interface. Hence the memory requirement on the embedded system for executing the test cases in this example is the memory requirement of the target monitor routine corresponding to the EIA-232 interface. From Table 5.4, this value is determined as approximately 1 KiByte. In the proposed approach in this thesis, the target monitor can be treated as an optimized software routine. Given its minimum memory requirement, it can also remain in the final production code. By this method, the developer/tester can make sure that the software that is subject to quality assurance is the same as the end-product, delivered to the customer.

This methodology introduced in the proposed approach provides a significant improvement over the existing MBT tools for executing model-based test cases in the embedded systems, in terms of the memory requirement on the embedded system. Existing tools such as [44], [43] generate the test harness (including test data, test drivers, stubs, etc) based on the size, complexity and the varying number of test cases in the testing process. This test harness is downloaded on the target to execute the test cases. In order to understand the memory requirement in the embedded system for the existing techniques in tools such as [43] and the significant advantage from the proposed approach, the following comparative study is carried out.

**Comparison with existing methodologies**

An existing MBT tool [43], which involves downloading the test harness in the embedded system is applied to four different embedded software application scenarios of increasing size and complexity. The percentage change in the memory overhead on the target w.r.t the application code-size is mea-

sured. In other words, the change in the memory requirement on the target w.r.t the application code-size, for executing the model-based test cases in the embedded system using the existing tool is measured. Similarly, for the proposed approach the percentage change in the memory overhead with

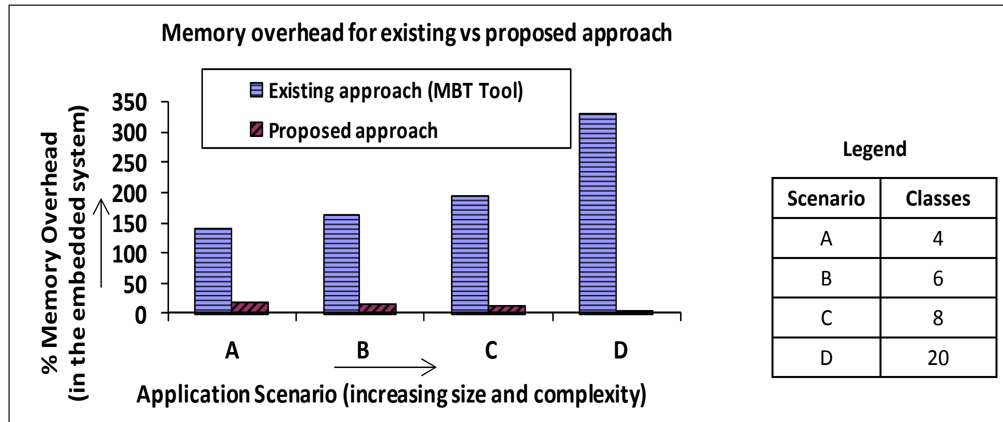



Figure 5.28: Memory overhead in an embedded system for existing vs proposed approach

reference to the application code-size on the embedded system for executing the model-based test cases is determined. This is carried out for the equivalent application scenarios on which the existing approach is applied. This comparison is illustrated in Figure 5.28.

Based on the results in Figure 5.28, it is clear that the existing technique involving source code instrumentation (in the MBT tool [43]) results in an increase in the memory overhead with an increasing application size and complexity. This increase in the memory overhead varies between 150% and 350% for the existing approaches, for executing model-based test cases in the embedded system. These results correspond to application scenarios A, B, C, and D comprising of 4, 6, 8 and 20 classes (in total), respectively in the design model. These scenarios correspond to examples from a LED toggling embedded software application (scenarios A, B), the stopwatch application (scenario C) and the MIDI system analyzer case study (scenario D). This memory overhead is significantly high and could show an exponential increase with an increase in the size and complexity of the application scenario. Moreover, the test harness that is downloaded on the target is removed after the testing process is completed and then the end-product is delivered. This necessarily means that the embedded software that is developed and subject to model-based quality assurance processes is not the same as the one delivered as the end-product.

On the other hand, the memory requirement (target monitor memory footprint-$S_{TM}$) for the proposed approach is constant, static and known beforehand. Note that the prototype target monitor requires only a total size of approximately 1 KiByte (Table 5.4) which is accommodative for resource constrained embedded systems. From Figure 5.28, it is evident that the percentage of the target monitor memory overhead is negligible in comparison with the application size. Since it remains constant and independent of the application's size and complexity, the percentage change in the memory overhead for the proposed approach shows a declining trend with reference to the increasing code-size of the application scenarios. For instance, from Figure 5.28 it is seen that the percentage change in the memory overhead for the proposed approach is approximately 17% to 2% (for the equivalent application scenarios).

Based on the runtime-time and memory overhead parameters of the target monitor, it can be stated that the proposed runtime monitoring mechanism for executing the model-based test cases in the embedded system is time and memory-aware. Hence, the overhead parameters can be included in the earlier phases of the embedded software development cycle (if necessary) and the target monitor can be included in the final production code.

# Chapter 6

# Summary and Outlook

This final chapter of the thesis provides a general summary and outlook of the work carried out in this thesis. The summary includes a discussion on the problems and challenges that have been introduced in this thesis and the significant advantages of the proposed approach (section 6.1). An outlook concerning future work related to the thesis is provided in section 6.2.

## 6.1 Summary

The manifold constraints in embedded software development such as the limited resources (e.g. memory), real-time requirements, growing variety and complexity and the stringent time-to-market constraints pose a big challenge for embedded software engineering. It is therefore essential that the development of embedded software systems take advantage of the new and automated methodologies such as MDA.

In the series of evolution in software engineering, MDA is considered as the next paradigm shift. Towards this direction, in the recent decade, the principles of MDA, i.e., MDD and MBT are gaining inroads individually for their applicability in embedded software engineering projects. On the other hand, the full-fledged and integrated usage of MDD and MBT in real-life embedded software engineering projects, for executing the model-based test cases in resource constrained embedded systems, is still an emerging technology.

In this context, based on an analysis of the currently existing model-based approaches for embedded systems, four main challenges are derived and addressed in this thesis. They are:

- Challenge (1): The proposed approach shall support the integrated usage of MDD and MBT (**integrated model-based approach**), thereby making use of corresponding modeling languages for the MDD and the MBT phases. This enables reusing the models from the MDD phase in the MBT phase towards the development of a test framework for executing the model-based test cases in the embedded systems.

  An example of using corresponding modeling languages for the MDD and MBT phases is the usage of UML for MDD and UML, UTP for MBT.

- Challenge (2): The test framework for executing the model-based test cases in the embedded system is generic i.e., its functionality and application are not restricted to a specific modeling

language (**generic test framework**). The generic notation and the test framework generation algorithm can be mapped to various modeling alternatives and tools for the automatic generation of the test framework.

- Challenge (3): The monitoring routine in the embedded system for executing the test input data is envisaged to be time and memory aware with bounded/measurable overhead parameters. This results in a less/minimally intrusive (**runtime/online software monitoring**) routine in the embedded system for executing the test cases.
- Challenge (4): The relevance and applicability of the proposed approach is envisaged to be demonstrated using a prototype implementation in real-life examples (**real-life embedded software engineering project example**).

Addressing challenges (1)-(4), this thesis proposes an integrated model-based approach and test framework for executing the model-based test cases in resource constrained embedded systems. The proposed approach, along with the contributions in this thesis is illustrated in Figure 6.1.

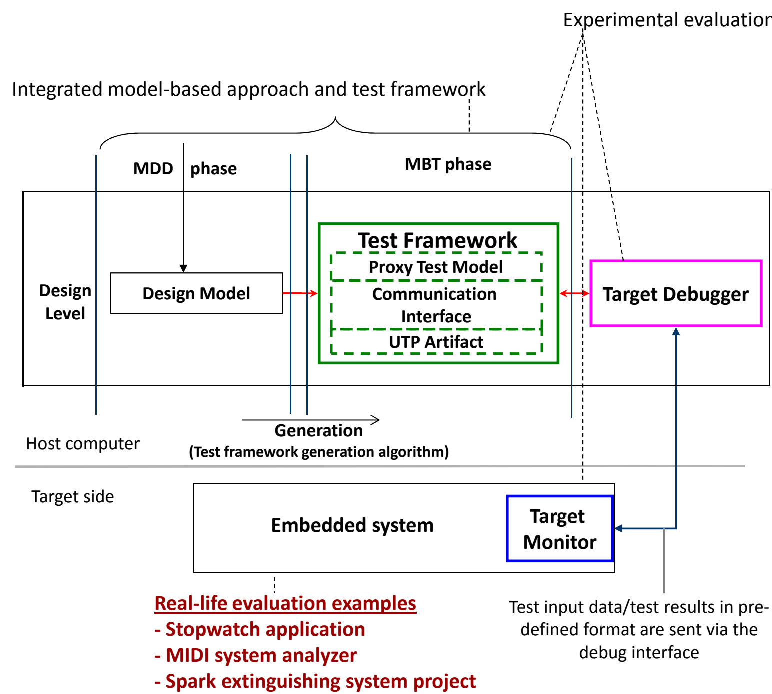



Figure 6.1: Proposed approach and contributions in this thesis

In the proposed approach, given a chosen SUT and the system design model (from the MDD phase), a test framework generation algorithm generates the test framework (during the MBT phase) at the host computer (Figure 6.1). The key idea of the test framework is to enable test automation and test case execution at the host computer. By this, the test harness for executing the model-based test cases is not downloaded on the target. Only the test input data is inserted and executed at the embedded system, using the target monitor. Corresponding (standardized) languages such as the UML and the UTP are used for the MDD and MBT phases respectively in the proposed approach. This relates to challenge (1) answering the question of how to develop an integrated model-based

approach and reuse models in the MBT phase from the MDD phase, for executing the model-based test cases in the embedded system.

The automatically generated test framework (comprising of the *proxy test model*, the *communication interface* and the *UTP artifact* (Figure 6.1)), uses two components namely a *target debugger* on the host and a runtime monitoring mechanism (denominated as the *target monitor*) on the embedded system for executing the model-based test cases. The target debugger at the host acts as a decoding and an information marshalling agent between the test framework and the target monitor. The *target monitor* at the embedded system is used to execute the test input data (received from the target monitor) and sends the trace data indicating the test results. Thus, the test input data from the test framework (to the target monitor) and the test results from the target monitor (to the test framework) are decoded and processed by the target debugger at the host (Figure 6.1), instead of the embedded system. With the aid of a decoding and information marshalling agent such as the target debugger on the host, significant overhead involved in interpreting the test data at the target is eliminated (challenge (3)). The test result is decoded at the host and displayed as UML diagrams (e.g. sequence diagrams) using the *proxy test model* component in the test framework (Figure 6.1).

In this thesis, a generic notation is introduced for the representation of the system design model and the test framework (challenge (2)). A test framework generation algorithm that makes use of the generic notation generates the necessary artifacts (i.e., the test framework) for executing the model-based test cases in the embedded system (Figure 6.1). The notation introduced for the system design model and the test framework are termed as generic, as the functionality and the application of the proposed approach is not restricted to a specific modeling language (challenge (2)). The generic notation and the test framework generation algorithm can be mapped to various modeling alternatives and tools for the automatic generation of the test framework. Whereas, in the prototype implementation of the proposed approach, UML is used for representing the system design model and UML, UTP are used for the representation of the components in the test framework.

To demonstrate the relevance and the applicability of the proposed approach, a prototype implementation of the test framework generation algorithm has been developed. Further, a prototype of the target debugger (at the host) and the target monitor (at the embedded system) are also developed for experimental evaluation. The prototype is evaluated on three real-life, resource constrained, embedded software engineering examples (challenge (4)), as seen in Figure 6.1.

An empirical evaluation is nevertheless incomplete without a complexity analysis of the proposed approach. Therefore, to gain insight into the complexity parameters of the proposed approach, theoretical estimates of the complexity measures involved in the test framework *generation* and the *runtime* complexity are analyzed. An experimental evaluation of the parameters in the complexity analysis is provided and compared with the theoretical estimates (challenges (3) and (4)). The *generation* time and memory complexity measures obtained by experiments on various scenarios indicate that the proposed approach has predictable time and memory requirements for generating the test framework. These results are inline with the inferences from the theoretical estimates for the complexity analysis. It is also evident that the time taken to generate the test framework for the chosen SUT increases with an increase in the complexity (e.g. number of classes, events) of the system design model. In the perspective of a software tester, the generation of the test framework for a chosen SUT using the proposed approach, can be considered as an one-time-activity. Therefore, this increase in the generation complexity measures on the host computer, may not be termed as a disadvantage.

Based on the *runtime*-time complexity measures, it is evident that the total time spent for executing the test cases in the embedded system using the proposed approach, comprises of only the time spent to decode the test stimuli by the monitor routine and execute the test stimuli in the embedded system. These temporal values, i.e., the time spent in the target monitor (e.g. for deciding, executing and sending trace data) is (static/constant and) measurable beforehand. For example, the time spent in the monitor to decode, execute and send an event consumed notification using the EIA-232 debug communication interface is 88.92 $\mu$s. Similarly, the only memory requirement (i.e., runtime-memory complexity) in the target (during runtime) for executing the model-based test cases in the target is that of the software-based runtime monitor in the target (e.g size approximately 1 KiByte). Therefore, the overhead parameters on the embedded system for executing the model-based test cases using the proposed approach are static/constant and measurable beforehand (challenge (3)).

A quantitative comparison on the percentage change in the memory overhead (i.e., *runtime*-memory complexity) for the existing approach and the proposed approach further provides insights on the scalability, applicability and superiority of the proposed approach over the existing approaches. The results indicate that the existing approaches in tools such as [43] introduces approximately 150% to 350% memory overhead in the embedded system for executing the test cases in simple application scenarios. In the proposed approach, the target monitor which is used to insert the test input data to the embedded system remains constant and independent of the number of test cases to be executed and their complexity. Therefore, the percentage change in the memory overhead for the proposed approach shows a declining trend with reference to the increasing code-size for the equivalent application scenarios (e.g. approximately 17% to 2%).

Thus, the proposed test framework approach constitutes a comprehensive quality assurance framework and test automation methodology, enabling the execution of high-level model-based test cases on the embedded system. In this test automation methodology, the test cases are executed at the host computer and only the test input data is executed on the target with the aid of the test framework and its components. Thus, the test framework approach provides the essential benefit of executing (online/offline) model-based tests in resource constrained embedded system without downloading the test harness on the embedded system. The performance metrics (i.e., the runtime complexity measurements) from the experimental evaluation reveals that the overhead parameters on the target are static/constant and measurable beforehand. This is a significant advantage and improvement over the existing methodologies, as the existing techniques in MBT tools (e.g. [43]) involve executing the model-based test cases more often on the host or with significant overhead (memory, performance) on the embedded system. Further, with the help of the test framework both functional and non-functional requirements can be verified, based on the trace data received from the target. Thus, in this thesis it is demonstrated that it is feasible to execute the test cases specified at higher abstraction levels (such as using UML sequence diagrams) in resource constrained embedded systems using the proposed test framework.

*These factors provide an important insight into the advantages (e.g. scalability, applicability, reliability) and superiority of the approach discussed in this thesis over the existing methodologies for executing model-based testing in embedded systems. This is a significant factor influencing the choice and applicability of a model-based approach and test automation, especially for resource-constrained embedded systems and real-life embedded software engineering projects.*

## 6.2 Outlook

What we see here is only the beginning of the potential uses of the proposed model-based test automation solution. There is still a plenty of research potential resulting from this thesis, in general. Some follow up aspects are described below.

- In the prototype implementation of the test framework approach discussed in this thesis, the test framework is primarily used to execute model-based unit-level and module-level test cases for one SUT. However, this approach can be extended to support more than one SUT and perform module-level integration testing covering several SUTs. Further system-level tests covering several modules (i.e., SUTs) may also be performed.

- Verification of temporal requirements with a maximum granularity of .1 $ms$ is demonstrated in this thesis. However, the timer granularity of the underlying RTOS determines the granularity with which the timing requirements can be specified in the test cases. Hence, for verifying temporal requirements of higher granularity (e.g. $\mu$s) using the proposed approach, a RTOS with higher granularity can be used.

- While monitoring/testing a system and acquiring the trace data one often faces the so-called "Heisenberg's effect": *Inspecting a system tends to influence the system's behavior* [29]. In case of embedded systems, this effect can be minimized if the microcontroller used in the embedded device supports real time trace functionality[1] [29]. This can be used together with the proposed software monitoring approach, i.e., hybrid monitoring approach to further minimize the instrumentation overhead. While the software-based runtime monitoring approach discussed in this thesis can be termed as a generic approach, hybrid monitoring approaches are more often hardware dependent.

  Thus, in general, it can be stated that the goal of a (software-based) monitoring technique used for applications such as model-based test case execution should be to minimize the overhead parameters. The overhead parameters should be measurable beforehand. The test harness or the additional artifact required to execute the model-based test cases on the target should be independent of the number of test cases to be executed and their complexity. By this, the bounded overhead parameters can be included at an earlier stage such as the system design phase or while specifying the test cases in the embedded software development cycle. By doing so, the influence on the real time characteristics of the embedded system due of the overhead introduced by the target monitor can be eliminated.

- The prototype of the test framework (on the host) developed in this thesis is implemented using a proprietary, industry-standard MDD/MBT tool, Rhapsody [42] [43]. Also, an industry-standard evaluation target platform [69] has been used during the experimental evaluation.

  On the other hand, the test framework approach can also be implemented using open-source MDD/UML modeling tools such as [80], [96] as the concepts in the proposed approach are generic in nature. Also, open-source target platforms such as [4] may be employed for an experimental evaluation of the proposed approach.

[1] Also referred as on-chip monitoring. A brief description of on-chip monitoring is provided in chapter 2.

Further, the generic notation for the system design model, and test framework can also be mapped to other modeling alternatives used in tools such as Matlab/Simulink [68] and LabView [60].

- The target information now available at the design level (i.e., at the target debugger) could be reused in the same manner as reusing software subroutines, such as to create test cases. Such test cases can be considered as an initial template which can be improved upon by the tester to create test cases manually. This provides the tester a certain level of independence from the MBT tools, especially when specifying the test cases manually. This can enhance test coverage, reduce specification gaps and lead to a more detailed specification without much additional effort. Such automatically generated test cases can also be used in conjunction with the test framework introduced in this thesis.

- Based on the model-based runtime monitoring solution, one could understand the energy characteristics of the embedded system in terms of energy monitoring and life energy prediction with the aid of energy diagrams. For instance, the target monitor can be used to send trace data pertaining to the energy characteristics (e.g. resource usage) of the embedded system. Such data on the host may be used (e.g. by the target debugger) to visualize and adapt (e.g. minimize) the energy characteristics/requirements of the target, in real time.

# Bibliography


[1] Altova UML Tool. http://www.altova.com/, last accessed: 2012.

[2] An introduction to MIDI. http://www.midi.org/aboutmidi/intromidi.pdf, 2012.

[3] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, pages 95–110, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Arduino: Open source electronics prototyping platform. http://www.arduino.cc/, 2012.

[5] Jan Axelson. *Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems*. Lakeview Research, 2nd edition, 2007.

[6] Paul Baker, Zhen Ru Dai, Jens Grabowski, Oystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer, Berlin, 2007.

[7] Paul Baker and Clive Jervis. Early UML Model Testing using TTCN-3 and the UML Testing Profile. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques*, TAICPART-MUTATION '07, pages 47–54, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Arnorld.S Berger. *Embedded Systems Design: An Introduction to Processes, Tools and Techniques*. CMP Books, 1st edition, 2001.

[9] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[10] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, A-MOST '07, pages 95–104, NY, USA, 2007. ACM.

[11] BridgePoint UML Tool. http://www.mentor.com/, 2012.

[12] C. Bunse, H.-G. Gross, and C. Peper. Applying a model-based approach for embedded system development. In *33rd EUROMICRO Conference on Software Engineering and Advanced Applications.*, August 2007.

[13] J.P. Calvez and O. Pasquier. Performance assessment of embedded Hw/Sw systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '95*, pages 52 –57, October 1995.

[14] CMX-RTX: Real-Time Operating System (RTOS). http://www.cmx.com/rtos.htm, 2012.

[15] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[16] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859 – 872, December 2004.

[17] S. Demathieu, F. Thomas, C. Andre, S. Gerard, and F. Terrier. First experiments using the UML profile for MARTE. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 50 –57, May 2008.

[18] Development of a central spark extinghusing system. http://www-05.ibm.com/de/events/innovate/pdf/Entwicklung-einer-GreCon-Funkenloeschanlage-fuer-die-Holzindustrie-mit-Rhapsody-von-der-Heide-Boerries.pdf, 2012.

[19] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, New York, NY, USA, 2007. ACM.

[20] Bruce Powel Douglass. *Real Time UML Workshop for Embedded Systems (Embedded Technology)*. Newnes, 2006.

[21] D. Dymek and L. Kotulski. Estimation of system workload time characteristic using UML timing diagrams. In *Third International Conference on Dependability of Computer Systems, DepCos-RELCOMEX*, pages 9 –14, June 2008.

[22] Embedded development tools . http://www.keil.com/, 2012.

[23] Enterprise Architect tool. http://www.sparxsystems.com/, 2012.

[24] Esterel programming language. http://www-sop.inria.fr/meije/esterel/esterel-eng.html, 2012.

[25] NModel framework. http://nmodel.codeplex.com/, 2012.

[26] R.B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using UML 2.0: promises and pitfalls. *Computer*, 39(2):59 – 66, February 2006.

[27] R. Fryer. Low and non-intrusive software instrumentation: a survey of requirements and methods. In *Proceedings on 17th IEEE Digital Avionics Systems Conference (DASC)*, volume 1, pages C22/1 –C22/8 vol.1, October 1998.

[28] Steve De Furia. *MIDI Book: Using MIDI and Related Interfaces.* Hal Leonard Publishing Corporation, 1987.

[29] Jack Ganssle. *The art of designing embedded systems.* Newnes, 2nd edition, 2008.

[30] Philipp Graf. eMote: A real-time Approach to Model-Based Testing of Embedded Software. http://www.model-based-testing.de/mbtuc11/presentations/Graf-FZI-eMOTE.pdf, 2012.

[31] Philipp Graf and Klaus D. Muller-Glaser. Gaining insight into executable models during runtime: Architecture and mappings. *IEEE Distributed Systems Online*, 8, March 2007.

[32] Philipp Graf, Klaus D. Muller-Glaser, and Clemens Reichmann. Nonintrusive Black- and White-Box Testing of Embedded Systems Software against UML Models. In *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, pages 130–138, Washington, DC, USA, 2007. IEEE Computer Society.

[33] GreCon GmbH. http://www.grecon.de, 2012.

[34] Gordon Ping Gu and Dorina C. Petriu. Early evaluation of software performance based on the UML performance profile. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 66–79. IBM Press, 2003.

[35] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Transactions on Software Engineering,*, 16(2):197 –211, February 1990.

[36] M. Harelick and A. Stoyen. Concepts from deadline non-intrusive monitoring. In *Frigeri, A.H., Halang, W.A., and Son, S.H. (Eds.):Proc. 24th IFAC/IFIP Workshop on Real-Time Programming (WRTP 99).* Elsevier, 30 May3 June 1999 1999.

[37] T. Harmon and R. Klefstad. Interactive back-annotation of worst-case execution time analysis for Java microprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2007.*, pages 209 –216, August 2007.

[38] Hartman, Alan. Model-Based Test Generation Tools. http://www.agedis.de, 2012.

[39] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Sung Deok Cha. Automatic test generation from statecharts using model checking. In *Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of BRICS Notes Series*, pages 15–30, 2001.

[40] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 327–341, London, UK, 2002. Springer-Verlag.

[41] Xiaowan Huang, Justin Seyster, Sean Callanan, Ketan Dixit, Radu Grosu, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. Software monitoring with controllable overhead. *Int. J. Softw. Tools Technol. Transf.*, 14(3):327–347, June 2012.

[42] IBM Rational Rhapsody Developer, Version 7.5. http://www.ibm.com, 2012.

[43] IBM Rational Rhapsody Test Conductor Add-on. http://www.btces.de/, 2012.

[44] IBM Rational Test RealTime. http://www-01.ibm.com/software/awdtools/test/realtime/, 2012.

[45] Padma Iyenghar. Test framework generation for model-based testing in embedded systems. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), 2011*, pages 267 –274, September 2011.

[46] Padma Iyenghar, Elke Pulvermueller, Clemens Westerkamp, Michael Uelschen, and Juergen Wuebbelmann. Model-based debugging of embedded software systems. *Gesellschaft Informatik (GI), softwaretechnik (SWT)*, 31-3, August 2011.

[47] Padma Iyenghar, Elke Pulvermueller, Clemens Westerkamp, and Juergen Wuebbelmann. Integrated model-based approach and test framework for embedded systems. In *Forum on Specification and Design Languages (FDL), 2011*, pages 1–8, sept. 2011.

[48] Padma Iyenghar, Michael Spieker, Pablo Tecker, Clemens Westerkamp, and Juergen Wuebbelmann. UML Target Animation: A Comparison of Debug Interfaces for Design Level Debugging. International conference on System, Software, SoC and Silicon Debug, S4D 2010. http://www.ecsi.org, September 2010.

[49] Padma Iyenghar, Clemens Westerkamp, Juergen Wuebbelmann, and Elke Pulvermueller. A model based approach for debugging embedded systems in real-time. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, NY, USA, 2010.

[50] Padma Iyenghar, Clemens Westerkamp, Juergen Wuebbelmann, and Elke Pulvermueller. Design level debugging of timing behavior in embedded systems: Using a model-based approach. In *9th IEEE International Conference on Industrial Informatics (INDIN), 2011*, pages 889 –894, july 2011.

[51] Yi Jiao, Kim Zhu, Qiang Yu, and Baifeng Wu. Towards model-driven methodology: a novel testing approach for collaborative embedded system design. In *10th International Conference on Computer Supported Cooperative Work in Design, 2006. CSCWD '06.*, pages 1 –5, May 2006.

[52] JTAG standard (IEEE 1149.1). http://standards.ieee.org/findstds/standard/1149.1-1990.html, 2012.

[53] Supaporn Kansomkeat and Wanchai Rivepiboon. Automated-generating test case using uml statechart diagrams. In *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, SAICSIT '03, pages 296–300, Republic of South Africa, 2003.

[54] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145 – 164, January 2003.

[55] H. Kashif, M. Mostafa, H. Shokry, and S. Hammad. Model-based embedded software development flow. In *4th International Design and Test Workshop (IDT)*, pages 1 –4, November 2009.

[56] S. Konrad, L.A. Campbell, and B.H.C. Cheng. Automated analysis of timing information in UML diagrams. In *19th International Conference on Automated Software Engineering, 2004. Proceedings.*, pages 350 – 357, September 2004.

[57] Leszek Kotulski and Dariusz Dymek. Using UML for the support of the test data generation. *Int. J. Crit. Comput.-Based Syst.*, 1:208–219, 2010.

[58] Padmanabhan Krishnan and Percy Pari-Salas. Model-Based Testing and the UML Testing Profile. In Jens Palsberg, editor, *Semantics and Algebraic Specification*, volume 5700 of *LNCS*, pages 315–328. Springer, 2009.

[59] Petri Kukkala, Jouni Riihimaki, Marko Hannikainen, Timo D. Hamalainen, and Klaus Kronlof. UML 2.0 Profile for Embedded System Design. In *Design, Automation and Test in Europe*, pages 710–715, 2005.

[60] LabVIEW System Design Software. http://www.ni.com/labview/, 2012.

[61] Beatriz Pérez Lamancha, Pedro Reales Mateo, Ignacio Rodríguez de Guzmán, Macario Polo Usaola, and Mario Piattini Velthius. Automated model-based testing using the uml testing profile and qvt. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '09, New York, NY, USA, 2009. ACM.

[62] Lauterbach-Microprocessor development tools. http://http://www.lauterbach.com/, 2012.

[63] Luciano Lavagno, Grant Martin, and Bran Selic, editors. *UML for real: design of embedded real-time systems*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[64] List of commecially available MBT tools. http://www.cs.waikato.ac.nz/research/mbt/Tools.pdf.

[65] Robert Love. *Linux kernel development*. Addison-Wesley Professional, 2010.

[66] C. MacNamee and D. Heffernan. Emerging on-ship debugging techniques for real-time embedded systems. *Computing Control Engineering Journal*, 11(6):295 –303, December 2000.

[67] Magic Draw UML Tool. http://www.magicdraw.com/, 2012.

[68] Matlab and Simulink. http://www.mathworks.com/, 2012.

[69] MCB1700 evaluation board. http://www.keil.com/mcb1700/, 2012.

[70] MCB2140 evaluation board. http://www.keil.com/mcb2140/, 2012.

[71] MERAPI UML modeling tool. http://www.willert.de/assets/Datenblaetter/UML-Getting-Started-Merapi-V1.0-en.pdf, last accessed: 2012.

[72] Kenneth L. Calvert Michael J. Donahoo. *TCP/IP Sockets in C, A Practical Guide for Programmers*. Morgan Kaufmann, 2nd edition, 2009.

[73] Michael Mlynarski, Baris Gueldali, Melanie Spaeth, and Gregor Engels. From design models to test models by means of test ideas. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '09, NY, USA, 2009. ACM.

[74] MSDN WIN32 library reference. http://msdn.microsoft.com/library/default.aspx, 2012.

[75] M. Mura, L.G. Murillo, and M. Prevostini. Model-based design space exploration for RTES with SysML and MARTE. In *Forum on Specification, Verification and Design Languages, 2008. FDL 2008.*, pages 203 –208, September 2008.

[76] P. V.R. Murthy, P. C. Anitha, M. Mahesh, and Rajesh Subramanyan. Test ready uml statechart models. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, SCESM '06, pages 75–82, New York, NY, USA, 2006. ACM.

[77] M. Mussa, S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj. A survey of model-driven testing techniques. In *9th International Conference on Quality Software, 2009. QSIC '09.*, pages 167 –172, August 2009.

[78] Gary J. Nutt. Tutorial: computer system monitors. *SIGMETRICS Perform. Eval. Rev.*, 5:41–51, January 1976.

[79] Object Management Group. http://www.omg.org, 2012.

[80] Papyrus UML Tool. http://www.papyrusuml.org/, 2012.

[81] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time Systems Symposium, 2008*, pages 481 –491, December 2008.

[82] Charles Petzold. *Programing Windows.* Microsoft press, 5th edition, 1998.

[83] Olaf Pfeiffer, Andrew Ayre, and Christian Keydel. *Embedded Networking with CAN and CANopen.* Annabooks, 2003.

[84] B. Plattner and J. Nievergelt. Special feature: Monitoring program execution: A survey. *Computer*, 14(11):76 –93, November 1981.

[85] Bernhard Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756 –764, November 1984.

[86] A. Pulka and A. Milik. VEST - an intelligent tool for timing SoCs verification using UML timing diagrams. In *Forum on Specification, Verification and Design Languages, 2008. FDL 2008.*, pages 118 –123, September 2008.

[87] Qt. User interface framework. http://qt.nokia.com/, 2012.

[88] Rational Rhapsody API Reference Manual. http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/rhapsody_api.pdf, 2012.

[89] Dale Rogerson. *Inside COM.* Microsoft Press, 1997.

[90] Uk Jeon Sang, Hong Jang Eui, and Bae Doo Hwan. Interaction-based behavior modeling of embedded software using UML 2.0. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC 2006.*, page 5 pp., April 2006.

[91] Beth A. Schroeder. On-line monitoring: A tutorial. *Computer*, 28:72–78, June 1995.

[92] Segger-Embedded software solutions. http://www.segger.com/, 2012.

[93] Chihhsiong Shih, Chien-Ting Wu, Cheng-Yao Lin, Pao-Ann Hsiung, Nien-Lin Hsueh, Chih-Hung Chang, Chorng-Shiuh Koong, and W.C. Chu. A model-driven multicore software development environment for embedded system. In *33rd Annual IEEE International on Computer Software and Applications Conference, COMPSAC '09.*, volume 2, pages 261 –268, July 2009.

[94] Mohammed El Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In *8th IEEE International Conference on Real-Time Computing Systems and Applications.*, 2002.

[95] Steve S. Skiena. *The Algorithm Design Manual.* Springer, 2008.

[96] Star UML-Open source UML/MDA platform. http://staruml.sourceforge.net/en/, 2012.

[97] A. Stefanescu, S. Wieczorek, and M.-F. Wendland. Using the UML Testing Profile for Enterprise Service Choreographies. In *36thEUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2010*, pages 12 –19, sept. 2010.

[98] Pietsch Stephan and Stanca-Kaposta Bogdan. Model-based testing with UTP and TTCN-3 and its application to HL7. In *http://www.testingtech.com*, 2008.

[99] Systems Modeling Language (SysML). http://www.sysml.org, 2012.

[100] L. Tan, J. Kim, O. Sokolsky, and I. Lee. Model-based testing and monitoring for hybrid embedded systems. In *Proceedings of the IEEE International Conference on Information Reuse and Integration, IRI 2004.*, pages 487 – 492, November 2004.

[101] TDS200C Oscilloscope. http://www.tek.com/oscilloscope/tds2000-digital-storage-oscilloscope, 2012.

[102] Texas Instruments. http://www.ti.com/, 2012.

[103] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *Parallel and Distributed Processing Symposium, 2003.*, April 2003.

[104] Thane.H. *Monitoring, testing and debugging distributed real-time systems.* PhD thesis, KTH Royal Institute of Technology, Sweden, 2000.

[105] The FreeRTOS project. http://www.freertos.org/, 2012.

[106] The GTK+ project. Multi-platform toolkit for creating GUIs. http://www.gtk.org/, 2012.

[107] Hideyuki Tokuda, Makoto Kotera, and Clifford E. Mercer. A real-time monitor for a distributed real-time operating system. *SIGPLAN Notices*, 24:68–77, November 1988.

[108] Visual Paradigm tool for UML. http://www.visual-paradigm.com/, 2012.

[109] J.J.P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897 –916, August 1990.

[110] ULINK family of Debug and Trace adapters. http://www.keil.com/ulink/, 2012.

[111] UML Testing Profile (UTP). http://utp.omg.org/, 2012.

[112] Unified Modeling Language (UML). http://www.uml.org/, 2012.

[113] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[114] VxWorks Operating System. http://www.windriver.com/products/vxworks/, 2012.

[115] G. Walters, E. King, R. Kessinger, and R. Fryer. Processor design and implementation for real-time testing of embedded systems. In *The AIAA/IEEE/SAE Conference on Digital Avionics Systems , 1998. Proceedings., 17th DASC.*, volume 1, November 1998.

[116] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *Software, IET*, 1(5):172 –179, October 2007.

[117] T. Westerlund, J. Paakkulainen, and J. Plosila. Back-annotation of timing information into a formal hardware model: a case study. In *International Symposium on Signals, Circuits and Systems, ISSCS 2005.*, volume 2, pages 625 – 628, July 2005.

[118] Willert Software Tools GmbH. http://www.willert.de/, 2012.

[119] Windows CE Operating System. http://msdn.microsoft.com/en-us/library/ms905511.aspx, 2012.

[120] wxWidgets. Cross-platform GUI library (C++). http://wxwidgets.org/, 2012.

[121] Liang Xi and Sun Xinxin. Research on the Mapping of UML2.0 Testing Profile to TTCN-3. In *2010 Third International Conference on Information and Computing (ICIC)*, volume 4, pages 280 –283, june 2010.

[122] J. Z-Nowicka. *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*. PhD thesis, Technische Universitaet Berlin, 2009.

[123] Z.R.Dai. *An approach to model-driven testing-Functional and real-time testing with UML2.0, U2TP and TTCN-3*. PhD thesis, Technische Universitaet Berlin, 2006.

# Publications

Parts of this work have been published in:

## Journals

(1) P. Iyenghar, E. Pulvermueller, C. Westerkamp, M. Uelschen, J. Wuebbelmann, *Model-Based Debugging of Embedded Software Systems*, Gesellschaft Informatik (GI) - Softwaretechnik (ST), August 2011 (31:3).

## **Conferences** (Peer-Reviewed)

(1) P. Iyenghar, M. Spieker, J. Wuebbelmann, C. Westerkamp, *A Generic Middleware for Automated Source Code-Level Coupling of Embedded Software-Subsystems Developed Using Heterogeneous Modeling Domains*, in 17th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2012, Krakow, Poland.

(2) M. Spieker, A. Noyer, P. Iyenghar, G. Bikker, J. Wuebbelmann, C. Westerkamp, *Model Based Debugging and Testing of Embedded Systems Without Affecting the Runtime Behaviour*, in 17th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2012, Krakow, Poland.

(3) P. Iyenghar, E. Pulvermueller, C. Westerkamp, J. Wuebbelmann, *Integrated Model-Based Approach and Test Framework for Embedded Systems*, in IEEE International Forum on Specification and Design Languages, FDL 2011, pages 1-8, Oldenburg, Germany.

(4) P. Iyenghar, E. Pulvermueller, C. Westerkamp *Towards Model-Based Test Automation for Embedded Systems Using UML and UTP*, in 16th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2011, pages 1-9, Toulouse, France.

(5) P. Iyenghar, *Test Framework Generation for Model-Based Testing in Embedded Systems*, in 37th IEEE International Euromicro conference on Software Engineering and Advanced Applications, SEAA 2011, pages 267-274, Oulu, Finland.

(6) P. Iyenghar, C. Westerkamp, J. Wuebbelmann, E. Pulvermueller, *Design Level Debugging of Timing Behavior in Embedded Systems: Using a Model-Based Approach*, in 9th IEEE International Conference on Industrial Informatics, INDIN 2011, pages 889-894, Lisbon, Portugal.

(7) P. Iyenghar, M. Spieker, P. Tecker, J. Wuebbelmann, C. Westerkamp, W.v.d. Heiden and A. Willert, *Applicability of an Integrated Model-Based Testing Approach for RTES*, in 9th IEEE International Conference on Industrial Informatics, INDIN 2011, pages 871-876, Lisbon, Portugal.

(8) P. Iyenghar, C. Westerkamp, J. Wuebbelmann, E. Pulvermueller, *A Model Based Approach for Debugging Embedded Systems in Realtime*, in 10th ACM international conference on embedded software, EMSOFT 2010, pages 69-78, NY, USA.

(9) P. Iyenghar, M. Spieker, P. Tecker, C. Westerkamp, J. Wuebbelmann, *UML Target Animation: A Comparison of Debug Interfaces for Design Level Debugging*, in System, Software, SoC and Silicon Debug conference, S4D 2010, Southampton, United Kingdom.

(10) P. Iyenghar, C. Westerkamp, J. Wuebbelmann, E. Pulvermueller, *An Architecture for Deploying Model Based Testing in Embedded Systems*, in IEEE International Forum on Specification & Design Languages, FDL 2010, pages 1-6, Southampton, United Kingdom.

## Workshops

(1) M. Spieker, P. Iyenghar, C. Westerkamp, J. Wuebbelmann, M. Corona *Entwicklung einer Middleware zur Target-spezifischen Codegenerierung aus unterschiedlichen Modelldomaenen*, Software Engineering fuer technische Systeme symposium - Forschungsnetz Industrial Informatics, 13.09.2012, Osnabrueck, Germany. (Poster & demo).

(2) P. Iyenghar, *Software Qualitaetssicherung fuer Geschaeftsprozesse mit eingebetteten Systemen*, in Protokoll-basierte Modellierung von Geschaeftsinteraktionen, PMBI 2011, Berlin, Germany. (Oral Presentation).

(3) P. Iyenghar, *Model-Based Testing of Deeply Embedded Systems*, Software Engineering fuer technische Systeme symposium - Forschungsnetz Industrial Informatics, 22.11.2011, Osnabrueck, Germany. (Poster & demo).

(4) P. Iyenghar, E. Pulvermueller, C. Westerkamp, M. Uelschen, J. Wuebbelmann, *Model-Based Debugging of Embedded Software Systems*, Development of Reliable Software Systems (Entwicklung zuverlaessiger Software-Systeme), 30-06-2011, Bosch Center-Feuerbach, Stuttgart, Germany (Oral Presentation).

(5) M. Spieker, P. Tecker, P. Iyenghar, E. Pulvermueller, C. Westerkamp, J. Wuebbelmann, *Fehlersuche im Dreivierteltakt Die Rhythmusanalyse des Klavierspiels*, TechnologieTag 2010, Osnabrueck, Germany. (Poster & demo).

(6) P. Iyenghar, M. Spieker, P. Tecker, *Embedded UML Target Debugger*, Software Engineering fuer technische Systeme symposium - Forschungsnetz Industrial Informatics, 15.09.2010, Osnabrueck, Germany. (Poster & demo).

(7) P. Iyenghar, M. Spieker, P. Tecker, *UML Target Animation- Backannotation Loesung fuer Design Level Debugging*, Software Engineering fuer technische Systeme, symposium - Forschungsnetz Industrial Informatics, 12.11.2009, Osnabrueck, Germany. (Poster & demo).

## Production Information/Tools

(1) E. Roemer, M. Matuschek, P. Iyenghar, M. Spieker, P. Tecker, W.v.d. Heiden *Embedded UML Target Debugger-Your Logic Analyzer for Embedded Software*, Willert Software Tools GmbH, http://www.willert.de. **Winner of the Embedded Award 2012** (category: Tools), at Embedded World 2012. *Press release:* `http://www.hs-osnabrueck.de/1029+M528a0e665f3.html`