View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Open Repository and Bibliography - Luxembourg

PhD-FSTC-2017-43
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defence held on 21/07/2017 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

CHUNHUI WANG
Born on 11 April 1987 in Harbin (Heilongjiang, China)

AUTOMATED REQUIREMENTS-DRIVEN TESTING OF EMBEDDED SYSTEMS BASED ON USE CASE SPECIFICATIONS AND TIMED AUTOMATA

DISSERTATION DEFENSE COMMITTEE

PROF. DR. ING. LIONEL BRIAND, Dissertation Supervisor
University of Luxembourg

DR. MEHRDAD SABETZADEH, Chairman
University of Luxembourg

DR. SHIVA NEJATI, Deputy Chairman
University of Luxembourg

PROF. DR.BRUNO LEGEARD, Member
Université de Franche-Comté

DR. GIOVANNI DENARO, Member
Università degli Studi di Milano

DR. FABRIZIO PASTORE, Advisory Member
University of Luxembourg

Abstract

The complexity of embedded software in safety-critical domains, such as automotive and avionics, has significantly increased over the years. For most embedded systems, standards require system testing to explicitly demonstrate that the software meets its functional and safety requirements. In these domains, system test cases are often manually derived from functional requirements in natural language plus other design artefacts, like UML statecharts. The definition of system test cases is therefore time-consuming and error-prone, especially given the quickly rising complexity of embedded systems.

The benefits of automatic test generation are widely acknowledged today but existing approaches often require behavioural models that tend to be complex and expensive to produce, and are thus often not part of development practice.

The work proposed in this dissertation focusses on the automated generation of test cases for testing the compliance between software and its functional and timing requirements. This dissertation is inspired by contexts where functional and timing requirements are expressed by means of use case specifications and timing automata, respectively. This is the development context of our industrial partner, IEE, an automotive company located in Luxembourg, who provided the case study used to validate the approach and tool described in this dissertation.

This dissertation presents five main contributions: (1) A set of guidelines for the definition of functional and timing requirements to enable the automated generation of system test cases. (2) A technique for the automated generation of functional test cases from requirements elicited in the form of use case specifications following a prescribed template and natural-language restrictions. (3) A technique that reuses the automatically generated functional test cases to generate timeliness test cases from minimal models of the timing requirements of the system. (4) A technique for the automated generation of oracles for non-deterministic systems whose specifications are expressed by means of timed automata. In the context of this dissertation, automated oracles for non-deterministic systems are necessary to evaluate the results of the generated timeliness test cases. (5) The evaluation of the applicability and effectiveness of the proposed guidelines and techniques on an industrial case study, a representative automotive embedded system developed by IEE.

Acknowledgements

First and foremost I would like to extend my sincere gratitude to my supervisor Professor Lionel Briand and co-supervisor Fabrizio Pastore. Without their patient guidance, the goal would not have been attained as smoothly, nor would the road have been as pleasant.

The research presented in this thesis was conducted under the collaboration between academia and industry. I would like to thank everyone who involved in this project that trying to improve the testing process in IEE S.A. for their commitment and patience, in particular, Thierry Stephany, Marcel Stolzenbach and David Wiseman.

I am grateful to the members of my defence committee: Mehrdad Sabetzadeh, Shiva Nejati, Bruno Legeard, and Giovanni Denaro for their insightful suggestions and remarks. I would like to additionally thank Prof. Bruno Legeard, and Dr. Giovanni Denaro for taking the time to travel to Luxembourg to attend my defence, despite their busy schedules.

I would like to also express my great thanks to all the friends that I have made in the tiny Grand Duchy of Luxembourg for the memorable moments that we have had. More specifically, I would like to thank all my colleagues in the Software Verification and Validation Lab for creating an inspiring and welcoming working environment.

Last but not least I would like to thank my parents in China for always supporting and believing in me. Without their support, this dissertation would not have been possible.

Chunhui WANG
At University of Luxembourg
On July 2017

Contents

Contents v

List of Figures viii

List of Tables xi

Acronyms xiii

1 Introduction 1
- 1.1 Context 1
- 1.2 Research Contributions 2
- 1.3 Organisation of the Dissertation 4

2 Requirement Analysis Guidelines for the Automated Generation of Functional and Timeliness Test Cases 5
- 2.1 Case Study System 6
  - 2.1.1 Testing of BodySense 6
- 2.2 Eliciting Functional Requirements 7
  - 2.2.1 Eliciting Use Cases with RUCM 7
  - 2.2.2 Domain Modelling 9
- 2.3 Modelling Timing Requirements 10
  - 2.3.1 Modelling the Timing Properties of Domain Entities 12
  - 2.3.2 Modelling the Timing Properties of the Environment 12

3 Automatic Generation of System Test Cases from Use Case Specifications 15
- 3.1 Motivation 16
- 3.2 Overview of the Approach 17
- 3.3 NLP Pipeline for UMTG 18
- 3.4 Evaluation of the Model Completeness 20
- 3.5 Identification of Constraints 21
- 3.6 Generation of the Use Case Test Model 21
- 3.7 Generation of Scenarios and Test Inputs 23
- 3.8 Generation of Test Cases 26
- 3.9 Empirical Evaluation 28
  - 3.9.1 RQ1. Is the modelling and analysis effort required by UMTG acceptable in an industrial context? 29

3.9.2 RQ2. Is UMTG effective in generating test cases that cover the requirements captured by use case specifications? . . . 30
3.9.3 RQ3. Does UMTG scale? . . . 30
3.9.4 Threats to Validity . . . 31
3.10 Conclusion . . . 32

4 System Testing of Timing Requirements based on Use Cases and Timed Automata 33
4.1 Limitations of Test Generation Based on Timed Automata . . . 34
4.2 Automated Testing with TAUC . . . 35
4.3 Overview of TAUC . . . 36
4.4 Identification of dependencies . . . 37
4.4.1 Analysis of attributes in object diagrams . . . 38
4.4.2 Identification of functional scenarios that trigger state transitions . . . 39
4.4.3 Identification of state invariants . . . 40
4.5 Generation of the Timeliness Test Model . . . 40
4.6 Identification of Timeliness Scenarios . . . 42
4.7 Generation of Executable Test Cases . . . 44
4.8 Empirical evaluation . . . 46
4.8.1 RQ1. Is TAUC effective in generating test cases able to detect faults that affect software timeliness? . . . 46
4.8.2 RQ2. Is the modelling and analysis effort required by TAUC acceptable in an industrial context? . . . 48
4.8.3 Threats to Validity . . . 48
4.9 Conclusion . . . 49

5 Oracles for Testing Software Timeliness with Uncertainty 51
5.1 Modelling of Test Cases . . . 53
5.2 Testing with UIO Sequences . . . 54
5.2.1 Generation of UIO sequences from TAs . . . 55
5.2.2 Consequences of uncertainty . . . 57
5.3 The *STUIOS* approach . . . 59
5.3.1 An Overview . . . 59
5.3.2 Probability Estimation with UPPAAL . . . 60
5.3.3 Generating Stochastic Test Cases . . . 60
5.3.4 Building PUIO Sequences . . . 63
5.3.5 Identifying Test Case Probabilities and Building Stochastic Test Cases . . . 64
5.3.6 Test Execution . . . 64
5.4 Empirical Evaluation . . . 66
5.4.1 RQ1. How does *STUIOS* compare with a simpler and less expensive alternative? 66
5.4.2 RQ2. Does *STUIOS* generate oracles that are effective in the presence of time uncertainty? . . . 67
5.4.3 RQ3. Can *STUIOS* negatively impact the efficiency of the testing process? . . 68
5.4.4 Threats to Validity . . . 69
5.4.4.1 Internal threats . . . 69
5.4.4.2 External threats . . . 69
5.5 Conclusions . . . 70

6 Related Work **71**
6.1 Generation of Functional System Test Cases . . . 71
6.2 Automated testing of software timeliness . . . 72
6.3 Definition of oracles for systems charachterized by time uncertainty . . . 73

7 Conclusions and Future Work **75**
7.1 Summary . . . 75
7.2 Future Work . . . 76

List of Papers **77**

Bibliography **79**

A Tool Suite Description **87**
A.1 UMTG Toolset . . . 87
A.1.1 Test Cases Generation with The UMTG toolset . . . 89
A.2 TAUC Toolset . . . 91
A.2.1 Test Generation Tools . . . 91
A.2.2 Empirical Evaluation Tools . . . 92
A.3 STUIOS Toolset . . . 93

List of Figures

2.1 Partial domain model for *BodySense* 10
2.2 Automaton that captures how TemperatureErrors are qualified and dequalified in *BodySense*. Edges are labelled by the triple guard (green), action (light blue), and update (blue). . . . 11
2.3 Environment automata for *BodySense* 13

3.1 Overview of the UMTG Approach 18
3.2 NLP pipeline applied to extract the behaviour of a Use Case 19
3.3 Part of the transducer that identifies constraints 19
3.4 Tags associated with the use case step in Lines 6 of Table 2.2 20
3.5 Portion of the domain model for *BodySense* 20
3.6 Metamodel for the Use Case Testing Model 22
3.7 The Use Case Testing Model derived from the Use Case in Table 4.1 23
3.8 GenerateInputs: the Test Generation Algorithm adopted in UMTG 24
3.9 Three scenarios built by UMTG for the use case in Table 2.2 26
3.10 Activities performed by UMTG to generate the test case in Table 3.3 28

4.1 Automaton that captures how TemperatureErrors are qualified and dequalified in *BodySense*. This is the same automaton shown in Figure 2.2. Edges are labelled by the triple guard (green), action (light blue), and update (blue). 34
4.2 Steps of TAUC. 37
4.3 Model of a functional scenario that covers the Alternative Flow 2.2 in Table 4.1. 38
4.4 Object diagram generated by *UMTG* to satisfy the path condition that covers the scenario in Figure 4.3. 39
4.5 Assignments required to exercise the scenario in Figure 4.3. 39
4.6 Automaton for the Scenario in Figure 4.3 41
4.7 Augmented Timing Requirements Automaton derived from the automaton in Figure 4.1. 42
4.8 Alignment of two Timeliness Scenarios 44
4.9 (a) Abstract Test Case generated from Timeliness Scenario 2, and (b) Executable Timeliness Test Case. 45

5.1 Automaton that captures how temperature errors are qualified and dequalified in *BodySense*. 54
5.2 A Test Case that checks if TemperatureErrors are qualified on time. 54
5.3 Automaton that shows a possible transition fault in the implementation of Figure5.1. . . 55
5.4 UIO sequence that characterises state DetectedQualified after the execution of the test case in Figure 5.2. 56
5.5 Test case of Figure 5.2 including the UIO sequence in Figure 5.4. 56
5.6 Test case that verifies the qualification of a temperature error that appears for a short time. 58

5.7 An overview of the *STUIOS* approach 59
5.8 The algorithm to generate stochastic test cases. 61
5.9 Test Automaton derived from the test case in Figure 5.7. 62
5.10 Portion of a faulty output sequence generated after the execution of the test case in Figure 5.6. 64

A.1 UMTG architecture (grey boxes show third party components, black boxes UMTG components with nested components) 87
A.2 Menus provided by the UMTG plug-ins 88
A.3 Example of an Abstract (left) and a corresponding Executable (right) Test Case Generated by UMTG 88
A.4 RCUM Syntax Report 89
A.5 Consistency Check Report 89
A.6 List of OCL Constraints 90
A.7 Mapping table 90
A.8 Traceability Links Generated for a Test Case. 90
A.9 Portion of a Mapping Table 91
A.10 Description of taucGen.jar 92
A.11 Description of muta.jar 92
A.12 Description of vmodels.jar 93
A.13 Description of verifier.jar 93

List of Tables

2.1 An example test case for *BodySense* 6
2.2 Use Case *Identify Initial Occupancy Status of a Seat* 8

3.1 An example test case for *BodySense* 16
3.2 Some constraints for the use case '*Identify Occupancy Status of a Seat*'. 21
3.3 A generated test case for *BodySense* 27
3.4 Mapping table for *BodySense* (excerpt) 27
3.5 Results obtained with the case study 29
3.6 Results obtained for RQ2 30
3.7 Results obtained for RQ3 31

4.1 *BodySense* Use Cases 35
4.2 Mapping Table for *BodySense* 46
4.3 Fault Coverage Measured Against 323 Faulty Versions. 47

Acronyms

API Application Programming Interface.

CTL Computation Tree Logic.

DXL Door eXtension Language.

EFSM Extended Finite State Machine.

FSM Finite State Machine.

ISO International Organization for Standardization.

LIN Local Interconnect Network.

NLP Natural Language Processing.

OCL Object Constraint Language.
OMG Object Management Group.

PUIO Probabilistic UIO sequences.

RFS Reference Flow Step.
RQ Research Question.
RUCM Restricted Use Case Modelling.

SMC Statistical Model Checking.
STUIOS Stochastic Testing with Unique Input Output sequences.
SUT System Under Test.

TAUC Test generation combining timed Automata and Use Case specifications.

UCS Use Case Specification.
UCTM Use Case Test Model.
UIO Unique Input Output sequence.
UML Unified Modelling Language.
UMTG Use case Modelling for system Tests Generation.

XMI XML Metadata Interchange.

Chapter 1

Introduction

1.1 Context

The complexity of embedded software in safety-critical domains, such as automotive and avionics, has significantly increased over the years. When testing critical systems in the automotive domain, similar to other application domains, rigorous testing processes conforming to established industry standards such as ISO 26262 [ISO, 2011] must be employed. Such standards mandate the level of rigour required for software testing, with the level of rigour increasing as the potential for the system to lead to loss of life increases. Even for systems that are deemed to have a relatively low probability of causing loss of life, ISO 26262 effectively mandates stringent requirement-based testing. In practice, this entails constructing system test cases that can be traced to high-level system requirements.

In current industrial practice, system test cases are often derived manually by the software engineers who read functional requirements written in natural language plus other software specification artefacts, for example timed automata or UML statecharts capturing the timing requirements of the system. The definition of system test cases is therefore time-consuming and error-prone, especially so given the quickly rising complexity of embedded systems in safety-critical domains.

This dissertation focusses on software development contexts where functional and timing requirements are expressed by means of use case specifications and timed automata, respectively. This is the development context of our industrial partner, IEE, an automotive company who provided the case study used to validate the techniques described in this dissertation, which is BodySense, an airbag control system. The specific development context considered in this dissertation has influenced the definition of the research problem addressed, and the identification of the solutions proposed. More specifically, this dissertation focuses on the problem of automatically generating test cases that verify the compliance of the system with both its functional requirements, i.e. *functional test cases*, and its timing requirements, i.e., *timeliness test cases*. We use the term software timeliness to indicate the ability of the software to satisfy timing constraints. We focus on functional and timeliness test cases because for companies developing embedded automotive systems like IEE, testing the compliance of the system with functional and timing requirements is of crucial importance for ensuring proper system behavior and safety. For an airbag control system, for example, the timely identification of error conditions and the proper execution of the consequent error management activities is necessary to guarantee the safety of the passengers of the car.

The benefits of automatic test case generation are widely acknowledged today and there are many proposed approaches in the literature [Escalona et al., 2011]. However most of the existing test automation approaches require very detailed behavioural models of the system in order to generate executable test cases. Usually, such detailed system models are expensive to produce, and their cost cannot be justified if they are used only for testing but not for other development activities. In case engineers decide to rely upon less detailed and high-level models, then they have to face the challenge that these models can be used only to generate abstract test cases. Abstract test cases can be concretized into executable test cases by means of test adaptation or test transformation approaches [Utting and Legeard, 2006], but the cost, however, for transforming an abstract test case into an executable test case is not negligible. The solutions proposed in this dissertation have been derived by the idea, inspired by our industrial partner, that a test generation technique can be successfully adopted in an industrial context if the additional information (e.g., models) required to enable test case generation is kept minimal.

Many approaches for the generation of functional test cases require that system specifications are captured as UML behavioural models such as activity diagrams [Linzhang et al., 2004], statecharts [Ryser and Glinz, 1999], and sequence diagrams [Nebut et al., 2006]. In modern industrial systems, these behavioural models tend to be complex and expensive if they are to be precise and complete enough to support test automation, and are thus often not part of development practice. To minimize the costs of modelling in this dissertation we propose solutions that exploit, to the maximum extent, the information available in common analysis and design artefacts, produced for other purposes such as contractual agreements with customers, to automate test case generation.

Concerning the testing of timing requirements, most of the existing techniques require that the timing constraints of the system are modelled by means of Timed Automata [Hessel et al., 2008, En-Nouaary et al., 2002, Aboutrab et al., 2013, En-Nouaary and Hamou-Lhadj, 2008] or UML statecharts [Mücke and Huhn, 2004]. However, to limit modelling effort, our observation is that software engineers derive very high level models that capture only the state transitions controlled by timing constraints. These models are thus used only to generate abstract test cases, which, as indicated above, lead to non-negligible additional costs if they are used to generate executable test cases. To automate the generation of executable test cases while keeping the modelling of timing requirements simple and high level, in this dissertation we define solutions that exploit the information available in use case specifications.

1.2 Research Contributions

In this dissertation, we address the problem of automatically generating functional and timeliness test cases from requirement specifications in contexts where development is use case-driven and additional modelling effort required for testing purposes must be kept to a minimum. To this end, we propose a set of guidelines for eliciting test-ready functional and timing requirements and a set of approaches that (1) generate functional test cases from requirements elicited by means of use case specifications, (2) combine the generated functional test cases with timing specifications to test timing requirements, (3) rely upon a stochastic solution to deal with the problem of building reliable oracles in the presence of partial state visibility and non-determinism. More precisely, we provide the following contributions:

1. **A set of practical guidelines (realistic modelling) for the definition of test-ready functional and timing requirements**. These guidelines include the adoption of a structured and analysable form of use case specifications, i.e., Restricted Use Case Modeling (RUCM) [Yue et al., 2013], and the adoption of Timed Automata [Alur and Dill, 1994] for eliciting timing requirements. RUCM is based on a template with restriction rules, enabling the automated extraction of behavioral information by reducing imprecision and incompleteness in use cases. Timed Automata are one of the formalisms usually adopted for the specification of the timing behaviour of the system.
   The proposed guidelines have been described in two conference papers [Wang et al., 2015, Wang et al., 2017] and are discussed in Chapter 2.
2. **A technique for generating executable system test cases by exploiting the behavioural information implicitly described in use case specifications**. We named this technique *Use Case Modelling for System Tests Generation (UMTG)*. UMTG requires a domain model of the system, which enables the definition of constraints that are used by UMTG to generate test data and oracles. UMTG avoids behavioral modelling by applying Natural Language Processing (NLP) to use case specifications expressed using RUCM. UMTG employs NLP to build Use Case Test Models (UCTMs) from RUCM specifications. A UCTM captures the control flow implicitly described in the RUCM specification. During NLP, a list of textual descriptions of pre, post and guard conditions in use cases is extracted. The software engineer further manually reformulates these textual descriptions using constraints written using the Object Constraint Language (OCL) [OMG, 2004] based on the domain model. UMTG combines UCTMs with the OCL constraints to enable automated test generation via constraint solving.
   This contribution has been published in a conference paper [Wang et al., 2015] and is discussed in Chapter 3.
3. **A technique for automatically generating executable test cases targeting software timeliness**. We named the technique *Test Generation combining Timed Automata and Use Case Specifications (TAUC)*. TAUC works by processing use case specifications and models of the timing requirements of the system elicited using timed automata to build the detailed models necessary to generate executable test cases. In this way, TAUC prevents software engineers from designing very detailed timed automata when use case specifications are available. In addition, TAUC automatically builds test suites that exercise the functionality regulated by timing requirements by optimising diversity among test inputs, thus increasing the probability of identifying problems dependent on specific input sequences.
   This contribution has been published in a conference paper [Wang et al., 2017] and is discussed in Chapter 4.
4. **A technique to automate the generation of effective oracles for timeliness test cases derived from timed automata in the presence of non-deterministic behaviour**. We named the technique *Stochastic Testing with Unique Input Output Sequences (STUIOS)*. STUIOS addresses the problem of generating oracles in the presence of non-deterministic behaviour. As main components of STUIOS, we introduce the concepts of *stochastic test cases* and *probabilistic UIO sequences (PUIO sequences)*. A PUIO sequence is an input-output sequence with an associated probability of observing the given output sequence in response to the inputs. The underlying idea is that probabilistic UIO sequences enable fault detection by determining if the output sequences observed through testing are unlikely, based on multiple executions of the same test cases. Stochastic test cases extend the same idea to the entire test case. A stochastic test case specifies the expected probability of observing a specific output sequence after a given

test input sequence; in addition, it includes a PUIO sequence that is used to check if the test execution brings the system to the expected state.
This contribution has been submitted and is under review. It is discussed in Chapter 5.

1.3 Organisation of the Dissertation

Chapter 2 describes a set of analysis and design guidelines that enable the adoption of the techniques proposed in this dissertation. We take advantage of the example models presented in this chapter also to provide an overview of *BodySense*, the case study system provided by IEE.

Chapter 3 describes UMTG along with the empirical results achieved with *BodySense*.

Chapter 4 describes TAUC along with the empirical results achieved with *BodySense*.

Chapter 5 describes STUIOS along with the empirical results achieved with *BodySense*.

Chapter 6 discusses related work.

Chapter 7 summarises the contributions of this dissertation and discusses perspectives on future work.

Appendix A provides a description of the software tools that were developed to support the proposed approaches and run empirical studies.

Chapter 2

Requirement Analysis Guidelines for the Automated Generation of Functional and Timeliness Test Cases

This dissertation presents a set of techniques that automate the generation of functional and timeliness test cases by relying upon the availability of analysis and design artefacts produced according to specific guidelines. The availability of these analysis and design artefacts is a precondition for the adoption of the techniques proposed in this dissertation. We intentionally rely upon artefacts that are commonly produced in industrial practice and that are realistic in terms of what can be expected from development engineers of embedded systems when explicit requirements and traceability between requirements and test cases are required. The industrial environment where our case study took place matches this description and helped us clarify our working assumptions.

In this chapter, we thus summarize the activities that should be performed by software engineers during analysis and design to produce the design artefacts processed by the techniques proposed in the dissertation. We propose two sets of guidelines: (1) functional requirements, and (2) timing requirements.

To give concrete examples of requirements defined according to the proposed guidelines we rely upon the analysis artefacts produced for *BodySense*, the case study system used to evaluate the techniques proposed in this dissertation. For this reason, this chapter presents also a brief overview of the *BodySense* system.

This Chapter proceeds as follows. Section 2.1 overviews the characteristics of *BodySense*, the case study system used to evaluate the techniques proposed in this dissertation. *BodySense* specifications are used to ease the description of the methodology itself. Section 2.2 describes the activities required to elicit functional requirements. Section 2.3 describes the activities that must be performed to elicit timing requirements.

2.1 Case Study System

BodySense is a capacitive sensing system supporting airbag suppression for child restraint systems in vehicles. Infant-only suppression, in combination with low-risk airbag deployment technology, is the most widely accepted safety strategy to fulfil the advanced airbag requirements of the FMVSS 208 regulation issued by the National Highway Traffic Safety Administration (NHTSA) in the U.S.

BodySense is able to distinguish between child restraint systems (class 1) and passengers in the front seat (class 2), suppressing airbag deployment in the first situation in case of a collision. The system also supports the seat belt reminder function and can generate an *airbag off* sign. Moreover, *BodySense* offers reliable classification regardless of seat belt tension, occupant position, weight or height. *BodySense* monitors a car seat for occupancy and classifies the occupant by using a sensor in the passenger seat. By measuring the change in the electrical field between an electrode and the vehicle body, *BodySense* is able to provide a robust classification between passengers and child restraint systems.

2.1.1 Testing of BodySense

BodySense is already in production and presents several critical functional and timing requirements that IEE engineers must verify in compliance with the ISO-26262 safety automotive standard [ISO, 2011].

BodySense is developed on a real-time operating system named MicroC/OS [Micrium Embedded Software, 2016] by using IBM Rational Rhapsody [IBM Rhapsody, 2017], a well-known modelling tool, for automated code generation in the C programming language. The system communicates with other components in the vehicle via LIN protocol. CANoe [Vector, 2017] is the software used for system testing, which simulates the sending of messages through the standard automotive LIN protocol.

To satisfy the requirements imposed by ISO-26262, IEE engineers must prove that each requirement has been tested and, to this end, traceability between software requirements and test cases is maintained. The tool adopted to keep traces of requirements and test cases is IBM Doors [IBM Doors, 2017], a requirements management tool, which is used by IEE engineers to both trace software requirements and the system test cases.

Table 2.1. An example test case for *BodySense*

Line	Operation	Inputs/Expectations
1	*Reset power and wait*	
2	ResetPower	Time=INIT_TIME
3	*Set occupant status - Adult*	
4	SetBus	Channel = RELAY Capacitance = 85
5	*Check SeatBeltReminder status is Occupied and AirBagControl status is Occupied*	
6	ReadAndCheckBus	D0=OCCUPIED D1=OCCUPIED

Table 2.1 gives a simplified version of a real test case for *BodySense*. The *BodySense* system test cases are stored in IBM Doors and include both high-level operation descriptions, i.e., informal descriptions of the operations to be performed on the system, and the concrete test driver operations that should be executed during testing. Lines 1, 3, and 5 provide high-level operation descriptions. These lines are followed by the name of the functions that should be executed by the test driver along with the corresponding input/output values. For instance, Line 4 invokes the function *SetBus* with a value indicating that the test driver should simulate the presence of an adult on the seat (for simplicity assume that the field sensor positioned on a seat sends the value 85 on the bus when an adult is seated).

2.2 Eliciting Functional Requirements

We assume that software engineers elicit requirements by means of use case specifications, and define domain models of the system by relying upon UML class diagrams. Use case specifications and domain models are common in requirements engineering practice [Larman, 2002], such as in IEE, the partner's organization in our research.

Further, we expect that use case specifications to be elicited according to a use case modelling approach named RUCM [Yue et al., 2013]. Previous work has shown that RUCM is usable and beneficial with respect to making use case specifications less ambiguous and more amenable to precise analysis and design. In addition to this, we stress that one of the results of the research work presented in this dissertation consists in the delivery of the RUCM method to our industrial partner, IEE. The software engineers at IEE have in fact started eliciting the requirements for their products by following the RUCM guidelines.

2.2.1 Eliciting Use Cases with RUCM

Restricted Use Case Modelling (RUCM) is a use case modelling approach [Yue et al., 2013] which is composed of a use case template that merges several aspects of existing templates and a set of well-defined restrictions to the use of natural language for documenting UCSs. RUCM aims to both reduce ambiguity and facilitate automated processing of textual requirements by mean of natural language processing.

The use case template of RUCM contains fields similar to those encountered in standard use case templates [Cockburn, 2000, Bittner, 2002, Phalp et al., 2007]. Table 2.2 shows a simplified version of the use case '*Identify Initial Occupancy Status of a Seat*' of *BodySense* written according to the RUCM rules.

Use case specifications written according to RUCM contain one basic flow (i.e., description of the main success scenario) and one or multiple alternative flows. The basic flow in Table 2.2 describes the main successful path that satisfies stakeholder interests. It contains a sequence of steps and a postcondition (Lines 3-10). A step can be one of the following interactions: an actor sends a request and/or data to the system (Line 4); the system validates a request and/or data (Line 6); the system replies to an actor with a result (Line 7); the system alters its internal state (Line 14).

Alternative flows describe other scenarios, both success and failure. An alternative flow always depends on a condition in a specific step of the basic flow. In RUCM, there are three types of alter-

Table 2.2. Use Case *Identify Initial Occupancy Status of a Seat*

1	**Precondition**
2	The system has been initialized
3	**1.1 Basic Flow**
4	1. The seat SENDS occupancy status TO the system.
5	2. INCLUDE USE CASE Classify occupancy status.
6	3. The system VALIDATES THAT the occupant class for airbag control is valid and the occupant class for seat belt reminder is valid.
7	4. The system SENDS the occupant class for airbag control TO AirbagControlUnit.
8	5. The system SENDS the occupant class for seat belt reminder TO SeatBeltControlUnit.
10	Postcondition: The occupant class for airbag control and the occupant class for seat belt reminder have been sent.
11	**1.2 Bounded Alternative Flow**
12	RFS 2-3
13	1. IF voltage error is detected THEN
14	2. The system resets classification filters.
15	3. ENDIF
16	4. RESUME STEP 1.
17	Postcondition: Classification filters have been reset.
18	**1.3 Specific Alternative Flow**
19	RFS 3
20	1. IF the occupant class for airbag control is not valid and the occupant class for seat belt reminder is not valid THEN
21	2. The system SENDS the previous occupant class for airbag control TO AirbagControlUnit.
22	3. The system SENDS the previous occupant class for seat belt reminder TO SeatBeltControlUnit.
23	5. RESUME STEP 1.
24	4. ENDIF
25	Postcondition: The previous occupant classes for airbag control and seat belt reminder has been sent to AirbagControlUnit and to SeatBeltControlUnit respectively.
26	**1.4 Specific Alternative Flow**
27	RFS 3
28	1. IF the occupant class for seat belt reminder is not valid THEN
29	2. The system SENDS the occupant class for airbag control TO AirbagControlUnit.
30	3. The system SENDS the previous occupant class for seat belt reminder TO SeatBeltControlUnit.
31	5. RESUME STEP 1.
32	4. ENDIF
33	Postcondition: The occupant class for airbag control has been sent to AirbagControlUnit and the previous occupant class for seat belt reminder has been sent to SeatBeltControlUnit.

native flows: *specific*, *bounded* and *global*. A specific alternative flow refers to a step in the basic flow (Lines 19 and 27). A bounded alternative flow refers to more than one step in the basic flow (Line 12) while a global alternative flow refers to any step in the basic flow. For specific and bounded alternative flows, the keyword '*RFS*' is used to refer to one or more reference flow steps (Lines 12, 19 and 27).

RUCM includes 26 rules that both restrict the use of natural language (16 rules) and enforce the use of keywords for specifying control structures (10 rules). Most of the rules in the former category (11 rules) aim to reduce ambiguity in use case specifications, while the latter category aims to facilitate the automated processing of the text written in use case specifications.

The keywords defined by RUCM specify conditional logic sentences (IF-THEN-ELSE-ELSEIF-ENDIF), concurrency sentences (MEANWHILE), condition checking sentences (VALIDATES THAT), and iteration sentences (DO-UNTIL). These keywords limit ambiguities in use case specifications. Keywords ABORT and RESUME STEP are used to describe an exceptional exit action and where an alternative flow merges back in its reference flow, respectively. An alternative flow ends either with ABORT or RESUME STEP, which means that the last step of the alternative flow should clearly specify whether the flow returns back to the reference flow and where (using keywords RESUME

STEP followed by a returning step number) or terminates (using keyword ABORT).

Some of the RUCM keywords are shown in the use case in Table 2.2. For example, the keyword '*INCLUDE USE CASE*' in Line 5 indicates the inclusion of other use cases.

The keyword '*VALIDATES THAT*' (Line 6) indicates a condition that must be true to proceed to the next step, otherwise an alternative flow is taken. In Table 2.2, the system proceeds to Step 4 (Line 7) only if the occupant classes for the airbag control and the seat belt reminder are valid (Line 6).

Extensions to RUCM

As result of our research, we have introduced three main extensions to the original RUCM template, described in the following.

According to the original RUCM template, bounded and global alternative flows begin with the keyword '*IF .. THEN*' for the guard condition under which the alternative flow is taken (Line 13). Instead specific alternative flows do not begin with the keyword '*IF .. THEN*' since a guard condition is already indicated in its reference flow step (Line 6). We have extended the RUCM template to handle the case in which composite conditions need to be further refined in multiple alternative flows. This is the case for Table 2.2 in which multiple alternative flows are considered to be executed (Lines 12, 19, and 27) when the condition in the reference flow step (Line 6) evaluates to false. This case was originally not covered by RUCM. Therefore, we suggest to use the '*IF .. THEN*' keyword also in specific alternative flows (Lines 20 and 28). The alternative flows are evaluated in the order they appear in the use case.

We have also introduced two other extensions to RUCM regarding the adoption of '*IF*' conditions and the way input/output messages are expressed. More specifically we have decided to follow the guidelines that suggest not to use multiple branches within the same use case path [Larman, 2002], thus enforcing the adoption of '*IF*' conditions only as a mean to specify guard conditions for alternative flows.

Also, we introduced the keyword '*SENDS .. TO*' as an RUCM extension for the system-actor interactions. The keyword eases automatic identification of steps for these interactions. According to our experience, in embedded systems the system-actor interactions are always specified in terms of messages. For instance, Step 1 in Table 2.2 (Line 4) indicates an input message from the seat to the system while Step 4 (Line 7) contains an output message from the system to the airbag. Additional keywords can be defined for other systems.

2.2.2 Domain Modelling

We expect that a domain model of the system is produced by relying upon UML class diagrams according to standard software engineering practice. In particular, we expect that the names appearing in entities and attributes of the domain model are consistent with the names appearing in the use case specifications. This practice is enforced by the test case generation technique presented in Chapter 3.

Figure 2.1 shows a portion of the domain model for *BodySense*. There are three car components: *BodySenseSystem, AirbagControlUnit, SeatBeltControlUnit*. *BodySenseSystem* periodically provides

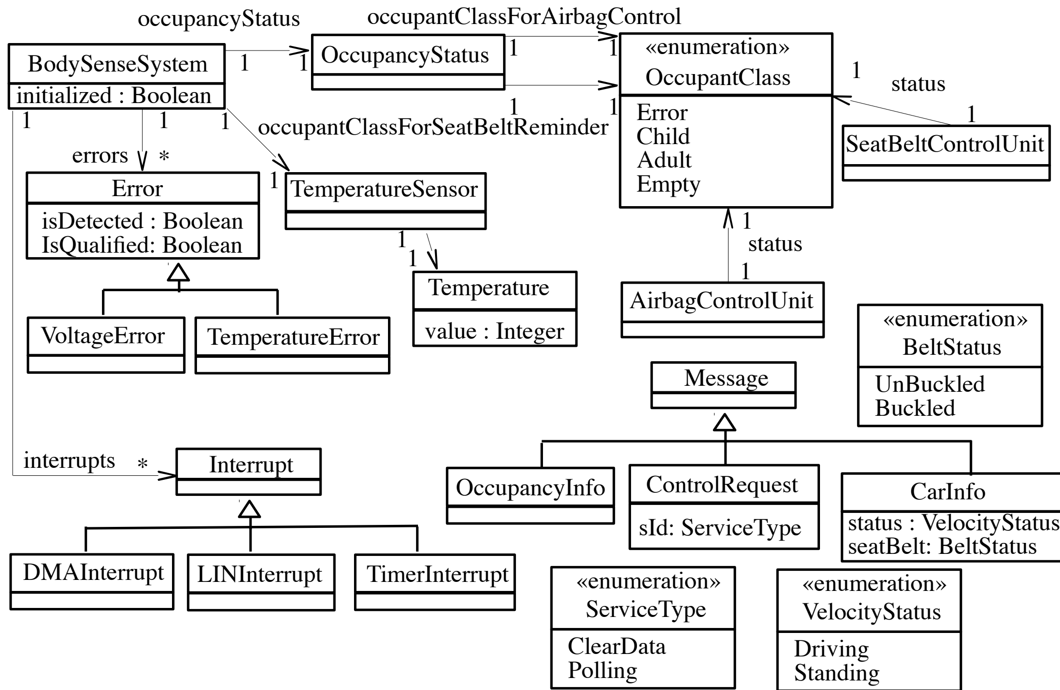


Figure 2.1. Partial domain model for *BodySense*

the classified occupancy status to the *AirbagControlUnit* and *SeatBeltControlUnit*. *BodySenseSystem* is connected to a temperature sensor which measures the environment temperature.

Figure 2.1 shows that the system must be able to deal with multiple types of errors. The cardinality in the association between *BodySenseSystem* and *Error* shows that zero or multiple errors can be detected in case the corresponding detection condition is satisfied. For example, if the environment temperature goes beyond a certain range, a temperate error will be detected.

Communication between *BodySenseSystem* and other car components, such as *AirbagControlUnit* or *SeatBeltControlUnit*, are via messages sending through the communication bus. For example, the message *OccupancyInfo* and *CarInfo* contain the data of occupancy status and seatbelt status required by *AirbagControlUnit* and *SeatBeltControlUnit*. Engineers (and mechanicians) can request the execution of a service by sending a *ControlRequest* message through the communication bus. For instance, the engineer can send a request for clearing the error data stored in the system by sending a *ControlRequest* message with a *ServiceId* (sId) of type *ClearData*.

BodySense also works with several types of interrupts (we show only two in this picture). For example, when sensor data is updated a *DMAInterrupt* is triggered, while when messages arrive on the communication bus a *LINInterrupt* is triggered.

2.3 Modelling Timing Requirements

In this dissertation, we assume that the timing requirements of a system are modelled using Communicating Timed Automata [Alur and Dill, 1994]. Timed automata are one of the formalisms commonly used to model timing properties of systems, along with UML statecharts [Larman, 2002]. In this dissertation, we rely upon the former primarily because of the availability of tools that enable model

based test generation and analysis of the timing properties. An example of this category of tools is the UPPAAL model checker [Bengtsson et al., 1995], which is one of the components integrated with our toolset. Approaches for translating UML statecharts into Timed Automata can be used when needed [David et al., 2002].

A timed automaton is a tuple $(L, l0, C, A, V, E, I)$, where L is a set of locations, $l0 \in L$ is the initial location, C is a set of clocks, A is a set of actions, V is a set of state variables, E is a set of edges between locations. I is a set of invariants assigned to locations. Each edge may have an action, a guard and a set of updates. Updates are expressed in form of assignments that can reset clocks or state variables. Each location might be associated with a state invariant that constrains clocks or state variables.

With communicating timed automata, the state of the system is captured by the values of state variables and the set of active locations across all the automata. Actions are used to synchronize different automata. Each action is expressed with the notation *event!* or *event?*. The notation *event!* indicates that the event is sent when the edge is fired, while the notation *event?* indicates that the edge is fired only if this event has been received from another automaton.

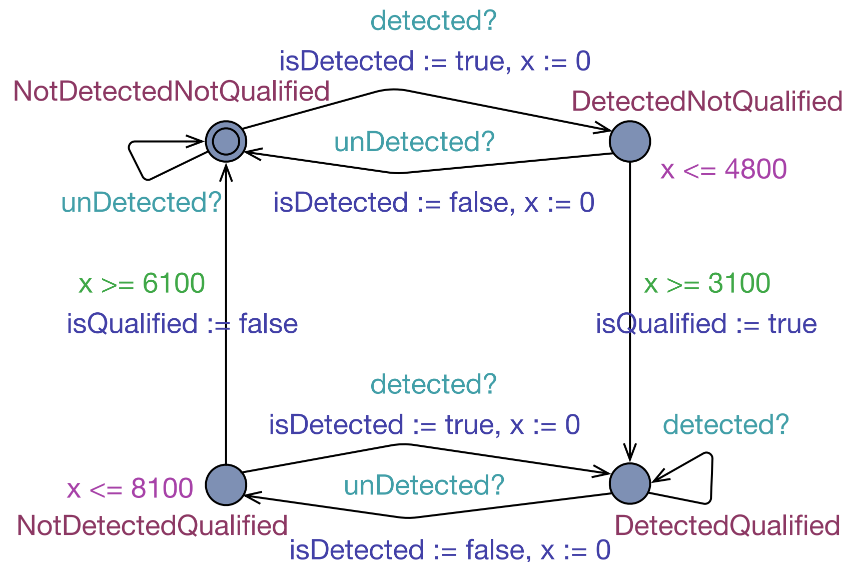


Figure 2.2. Automaton that captures how TemperatureErrors are qualified and dequalified in *BodySense*. Edges are labelled by the triple guard (green), action (light blue), and update (blue).

Figure 2.2 shows a timed automaton that captures the timing properties concerning the qualification of temperature errors. The variable x is a clock variable, while *isDetected* and *isQualified* are two state variables. The edge that connects locations *NotDetectedNotQualified* and *DetectedNot-Qualified* can be fired only when the event *detected*, which indicates that a temperature error has been detected, has been received by the automaton. The location *DetectedNotQualified* has an invariant that indicates that the clock variable x must be below 4800 ms when the location is active. The guard condition on the edge between the location *DetectedNotQualified* and *DetectedQualified* indicates that this edge cannot be fired if the clock x is below 3100. In effect, the edge between locations *DetectedNotQualified* and *DetectedQualified* can be fired anytime when the clock variable x has a value between 3100 and 4800. TAs are usually modeled this way (i.e., by including time ranges) in practice, to allow for timing uncertainties, since software engineers cannot predict/control with abso-

lute precision when operations start/end when software is running on the actual deployment platform. In Chapter 5 we discuss the problems caused by this uncertainty in the timing specifications of the system.

In general, we expect that software engineers use timed automata to capture both the timing properties of the entities of the domain model and the properties of the environment. The following paragraphs provide additional details for these two cases.

2.3.1 Modelling the Timing Properties of Domain Entities

The identification of timing requirements is a manual activity. We expect that software engineers define a timed automaton for each entity in the domain model that features one or more timing constraints. In the rest of the dissertation, we use the term *timing requirements automata* to refer to timed automata used to model the timing properties of domain entities.

State variables appearing in the timed automata have counterparts in the domain model. More specifically, we expect that each state variable is also an attribute of the entity modelled by the timed automaton. Figure 2.2 shows the automaton that captures the timing constraints related to the entity *TemperatureError*. The variables *isDetected* and *isQualified* are two attributes of the entity *TemperatureError* and, therefore, the assignments to the state variables *isDetected* and *isQualified* capture changes to the state of the entity *TemperatureError*.

Events in the network of timed automata are used both to synchronize with different timed automata, and to indicate the processing of an input or the generation of an output. We have observed that in practice, software engineers may also use events to avoid details that are unnecessary to describe timing requirements, i.e. details about the functional operations that trigger certain state transitions. In fact, software engineers may produce timed automata with edges synchronized with events that stand for the execution of one or several specific functional scenarios. These events are not generated by any other automata of the system; we call these events *scenario events*, since they are expected to be generated as a result of the completion of a functional scenario.

To enable automated generation of timeliness test cases we constrain the adoption of scenario events by requiring that scenario events always lead to updates of state variables (the scenario event *detected* in Figure 2.2, for example, updates the state variable *isDetected*). Proper updating of state variables is required since it is the analysis of such updates that enables one of the techniques proposed in this dissertation, TAUC, to identify dependencies between use case specifications and timed automata (see Chapter 4).

Chapter 4.4 shows how TAUC automatically detects *scenario events* without requiring additional information from software engineers. Our modelling approach allows software engineers to follow usual design practices with reasonably small timed automata models.

2.3.2 Modelling the Timing Properties of the Environment

Environment automata are used to capture the arrival frequency of inputs, including interrupts or messages. This information is necessary to avoid the generation of invalid test cases, e.g. test cases that send inputs with a frequency not supported by the system bus.

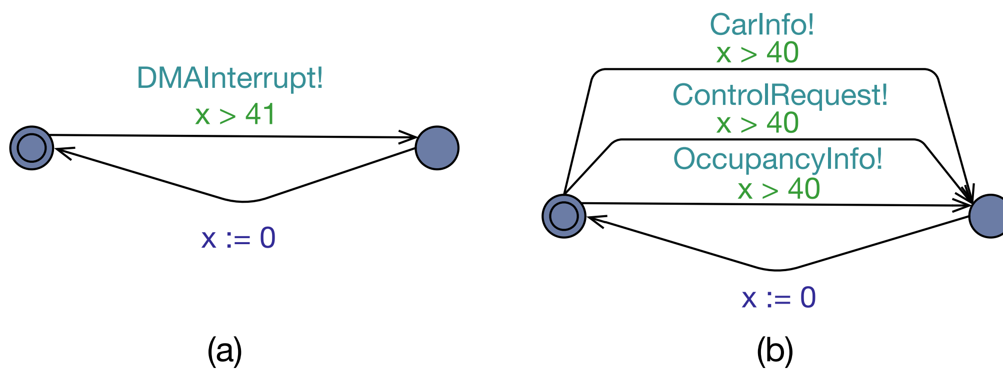


Figure 2.3. Environment automata for *BodySense*

We model the arrival of inputs by means of events. The event names must match entity names used in the domain model. Figure 2.3 shows two environment automata of *BodySense* that capture timing characteristics of interrupts and messages, respectively. For example, the minimal interarrival time of DMA interrupts is 41 milliseconds (a), while messages' minimal interarrival time is 40 milliseconds (b).

Chapter 3

Automatic Generation of System Test Cases from Use Case Specifications

In this chapter, we present *Use Case Modelling for System Tests Generation (UMTG)*, an approach for the generation of executable functional test cases. UMTG requires use case specifications written according to RUCM and a domain model of the system, which enables the definition of constraints that are used by UMTG to generate test data and oracles. In addition, the approach includes a specification refinement activity during which engineers are asked to provide constraints, referring to the domain model, defined with the Object Constraint Language (OCL) [OMG, 2004]. OCL is the natural choice when defining high-level constraints on class diagrams. The constraints are used to generate test data via constraint solving [Ali et al., 2013].

UMTG employs NLP to build *Use Case Test Models (UCTM)* from RUCM specifications. UCTMs capture the control flow implicitly described in the RUCM specification. During NLP, a list of textual descriptions of pre, post and guard conditions in use cases is extracted. The software engineer further manually reformulates these textual descriptions using OCL constraints based on the domain model, iteratively refining the latter when required. Our approach combines UCTMs with the OCL constraints to enable automated test generation. As a final step, it automatically generates system test cases with test inputs and outputs from UCTMs via constraint solving [Ali et al., 2013].

To summarise, the contributions of this chapter are:

- UMTG, an approach for the automatic generation of executable system test cases from use case specifications and a domain model, without resorting to behavioral modelling;
- an NLP technique generating test models (UCTMs) from use case specifications expressed with RUCM;
- an algorithm combining UCTMs and constraint solving to automatically generate test inputs;
- a demonstration of the feasibility of the approach based on an industrial case study in the automotive domain.

This chapter is structured as follows. Section 3.1 introduces the challenges that need to be tackled in order to design an automated test generation approach for embedded systems. In Section 3.2, we provide an overview of UMTG. In Section 3.3 we present the NLP pipeline in our approach. In Sections 3.4 to 3.8, we detail the automated activities in UMTG. In Section 3.9, we present our empirical evaluation.

3.1 Motivation

This section provides details about the challenges that need to be tackled in order to automate the generation of functional test cases in the context of an embedded system.

To illustrate the problem we introduce a simplified version of a test case for a scenario of the use case '*Identify initial occupancy status of a seat*' for *BodySense* in Table 3.1. Lines 1, 3, and 5 provide high-level operation descriptions, i.e., informal descriptions of the operations to be performed on the system. These lines are followed by the name of the functions that should be executed by the test driver along with the corresponding input/output values. For instance, Line 4 invokes the function *SetBus* with a value indicating that the test driver should simulate the presence of an adult on the seat (for simplicity assume that the field sensor positioned on a seat sends the value 85 on the bus when an adult is seated).

Table 3.1. An example test case for *BodySense*

Line No.	Operation	Inputs/Expectations
1	*Reset power and wait*	
2	ResetPower	Time=INIT_TIME
3	*Set occupant status - Adult*	
4	SetBus	Channel = RELAY Capacitance = 85
5	*Check SeatBeltReminder status is Occupied and AirBagControl status is Occupied*	
6	ReadAndCheckBus	D0=OCCUPIED D1=OCCUPIED

The exhaustive test cases needed to validate a safety-critical embedded system are difficult both to derive and maintain as requirements are changing. For instance, the real test case from which we derived the simplified test case in Table 3.1, includes 30 test steps, around 50 variable assignments, and 15 references to other specifications providing further explanations about the inputs. The effort required to specify test cases for the whole *BodySense* is overwhelming. A test suite exercising all use case scenarios would include more than 400 such test cases. Without automated test case generation, such testing activity remains not only expensive but also infeasible under typical time constraints.

Within the context of testing safety-critical embedded software system such as *BodySense*, we identify three challenges that need to be considered for the automatic generation of system test cases from functional requirements:

Feasible Modelling. Most of the existing automatic system test generation approaches are model-based and rely upon behavioural models such as sequence or activity diagrams. In complex industrial systems, behavioural models that are precise enough to enable test automation are quite complex that their specification cost is prohibitive and the task is often perceived as overwhelming by engineers. To evaluate the applicability of behavioral modelling on *BodySense*, we asked the IEE software engineers to specify system sequence diagrams corresponding to the use cases of *BodySense*. For example, the system sequence diagram for the use case '*Identify initial occupancy status of a seat*', includes 74 messages, 19 nested blocks, and 24 references to other system sequence diagrams that had to be derived. This was clearly considered too complex for software engineers and required significant help from the authors of this paper, and many iterations and meetings. Note that this is a relatively simple

use case compared to what can be found in other, more complex systems at IEE, and that, therefore, our conclusion was that behavioural modelling was not a practical option for test automation.

Test Data Generation. Without behavioral modelling, most approaches mainly exploit NL requirements specifications in which it is hard to extract test data for executable test cases. Test cases derived in such a way that requires significant manual intervention. For instance, even the simplified test case in Table 3.1 has 50 variable assignments to be manually entered as test input. Automatically generating test data, and not just abstract test scenarios, is therefore important.

Deployment of the Software under Test. Execution of test cases for a system like *BodySense* entails the deployment of software under test on the target environment. To speed up software testing, test case execution is typically automated through test scripts invoking test driver functions. These functions simulate sensor inputs and computational results receiving from a communication bus. Any test generation approach should generate appropriate functions and test data in a processable format for the test driver. For instance, the test drivers in *BodySense* need to invoke driver functions (e.g., *SetBus*) to simulate occupancy on a seat.

3.2 Overview of the Approach

Figure 3.1 shows the main steps of the approach. The goal of UMTG is to address the challenges given in Section 3.1. In UMTG, behavioral information and high-level operation descriptions are generated from use cases (*the first challenge*), test inputs are generated through constraint solving (*the second challenge*), while test driver functions corresponding to informal descriptions and oracles implementing the postconditions of the use case scenarios are generated through mapping tables provided by the software engineer (*the third challenge*).

The software engineer elicits requirements with RUCM (Step 1). The domain model is manually created as a UML class diagram (Step 2). UMTG automatically checks if the domain model includes all entities mentioned in the use cases (Step 3) and tool support is provided to guide engineers in completing the domain model. NLP is used to extract domain entities from the use cases. Missing entities are shown to the software engineer who refines the domain model (Step 4). Steps 3 and 4 are iterative: the domain model is refined until it is complete.

Once the domain model is completed, textual descriptions of pre, post and guard conditions in the use cases are automatically extracted (Step 5) to be reformulated as OCL constraints by engineers (Step 6). UMTG further processes the use cases with the OCL constraints to generate a Use Case Test Model for each use case (Step 7). A generated test model is a directed graph that explicitly captures the implicit behavioural information in the corresponding use case.

UMTG relies on constraint solving for OCL constraints that are attached to the nodes of the test models. The goal is to generate test inputs associated with use case scenarios (Step 8). We use the term use case scenario for a sequence of use case steps that start with a use case precondition and ends with a postcondition of either a basic or alternative flow. Test inputs cover all paths in the testing model and therefore cover all possible use case scenarios.

The software engineer provides a mapping table that maps high-level operation descriptions and

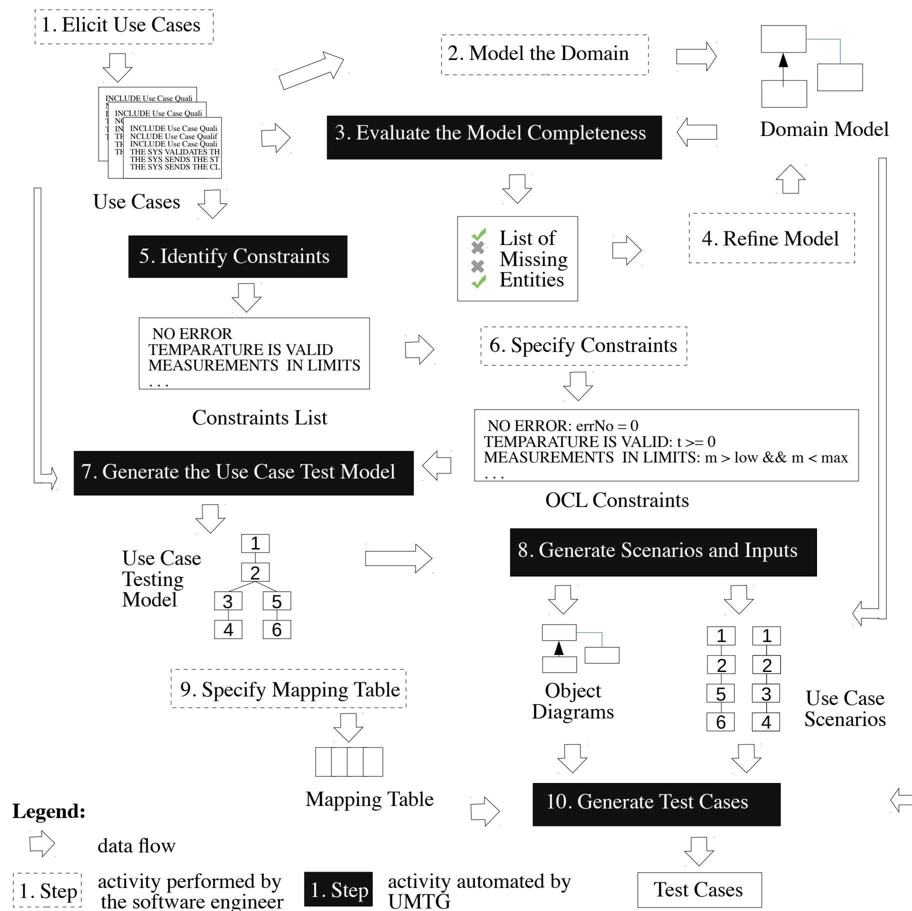


Figure 3.1. Overview of the UMTG Approach

test inputs to the concrete driver functions and inputs that should be executed by test cases (Step 9). Executable test cases are automatically generated through the mapping table (Step 10). If the test infrastructure and hardware drivers change in the course of the system lifespan, then only the mapping table needs to change.

The rest of the chapter provides a detailed description of each step of UMTG shown in Figure 3.1, except Steps 1 and 2 already introduced in Chapter 2.2.

3.3 NLP Pipeline for UMTG

Three UMTG steps in Figure 3.1, 'evaluate the model completeness', 'identify constraints', and 'generate the use case test model', are supported by an NLP application to extract behavioral information from RUCM use cases.

The NLP application in UMTG is based on the GATE workbench [Cunningham et al., 2002], an open source NLP framework, and implements the analysis pipeline in Figure 3.2. The pipeline includes both default NLP components (grey) and components configured to process RUCM use cases (white). The *Tokenizer* splits the use cases into tokens. The *Gazetteer* identifies the RUCM keywords. The *POS Tagger* tags tokens according to their nature: *verb*, *noun*, and *pronoun*. The

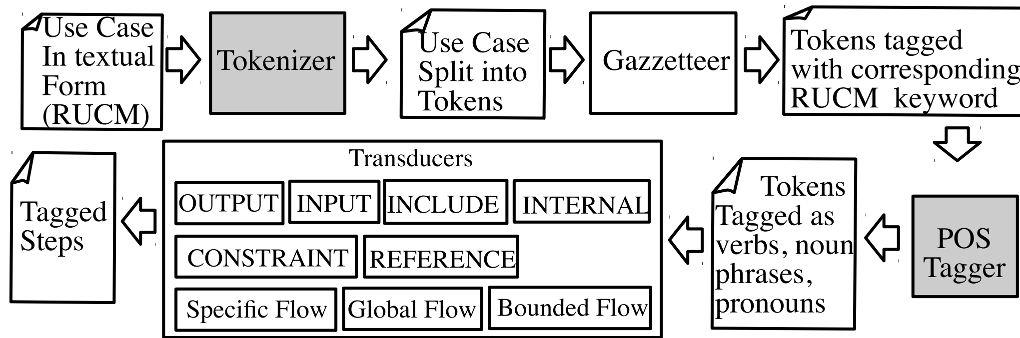


Figure 3.2. NLP pipeline applied to extract the behaviour of a Use Case

pipeline is terminated by a set of *transducers* that tag block of words to distinguish the kinds of RUCM steps: output, input, include, and internal operations. The pipeline has also *transducers* to distinguish specific, bounded and global alternative flows, plus reference blocks, and constraints part of guard and post conditions.

Figure 3.3 gives an example transducer for constraints. Capital names on the arrows correspond to the transducer's inputs, i.e., tags previously identified by either the POS tagger, the gazzetteer or other transducers. Italic names show the tags associated with the transducer to the words corresponding to the transducer input. Figure 3.4 gives the tags associated with the use case step in Line 6 of Table 2.2 after the execution of the transducer in Figure 3.3. In Figure 3.4, multiple tags are associated with the same block of words: the clause '*the occupant class for airbag control is valid*' is tagged both as a *simple constraint* and as a part of a *composite constraint*.

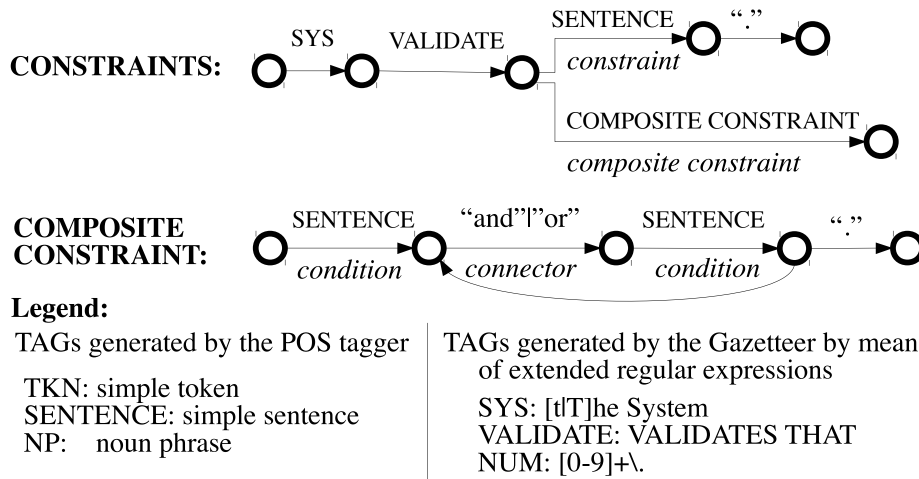


Figure 3.3. Part of the transducer that identifies constraints

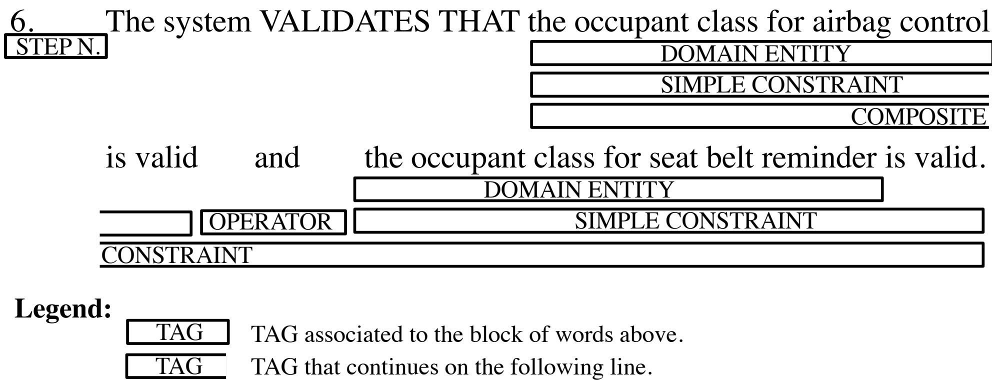


Figure 3.4. Tags associated with the use case step in Lines 6 of Table 2.2

3.4 Evaluation of the Model Completeness

UMTG automatically evaluates the model completeness (Step 3 in Figure 3.1) by identifying missing domain entities. This is done by checking correspondences between the domain entities identified by the NLP application and the entities in the domain model. Please recall that we assume naming conventions for the use cases and domain model[1].

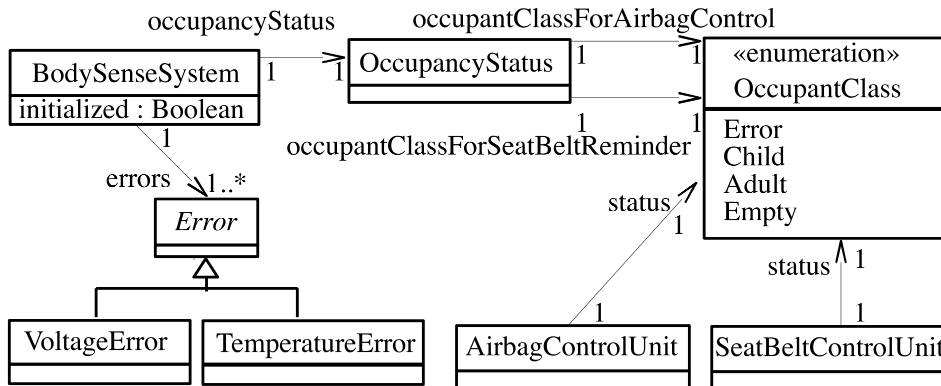


Figure 3.5. Portion of the domain model for *BodySense*

Sometimes domain entities identified in a use case are not modelled as classes but as attributes. Figure 3.5 shows a simplified portion of the domain model for *BodySense* where the domain entities '*occupant class for airbag control*' and '*occupant class for seat belt reminder*' are modelled as attributes of the class *OccupancyStatus*. UMTG follows a simple yet effective solution to check class and attribute names. For each domain entity identified through NLP, UMTG generates an entity name by removing all white spaces and by putting all first letters following white spaces in capital. For instance, the domain entity '*occupant class for airbag control*' becomes '*OccupantClassForAirbagControl*'. UMTG checks generated entity names in the domain model. If the entity name appears either as a class name or as an attribute name, the entity is considered present, otherwise it is considered missing.

[1]The precise identification of correspondences between the domain entities in the use cases and the entities in the domain model in the presence of inconsistent name usage can be tackled by syntactic and semantic similarity checking algorithms but it is not in the scope of this dissertation.

Table 3.2. Some constraints for the use case '*Identify Occupancy Status of a Seat*'.

#	Condition in the Use Case	Corresponding OCL Constraint
1	the occupant class for airbag control is valid	$BodySenseSystem.allInstances()$ $\rightarrow forAll(b \mid b.occupancyStatus.occupantClassForAirbagControl <> OccupantClass :: Error)$
2	the occupant class for seat belt reminder is valid	$BodySenseSystem.allInstances()$ $\rightarrow forAll(b \mid b.occupancyStatus.occupantClassForSeatBeltReminder <> OccupantClass :: Error)$
3	voltage error is detected	$Error.allInstances()$ $\rightarrow select(e \mid e.oclIsKindOf(VoltageError)) \rightarrow forAll(v \mid v.valid = true)$
4	the occupant classes for airbag control and seat belt reminder have been sent	$AirbagControlUnit.allInstances() \rightarrow forAll(a \mid a.status = a.bodySense.occupancyStatus$ $.occupantClassForAirbagControl)\ AND\ SeatBeltControlUnit.allInstances()$ $\rightarrow forAll(s \mid s.status = s.bodySense.occupancyStatus.occupantClassForSeatBeltReminder)$
5	The system has been initialized	$BodySenseSystem.allInstances() \rightarrow forAll(b \mid b.initialized = true)$
6	The system resets classification filters	$BodySenseSystem.allInstances() \rightarrow forAll(b \mid b.occupancyStatus.occupant$ $ClassForSeatBeltReminder = null\ AND\ b.occupancyStatus.occupantClassForAirbagControl = null)$

3.5 Identification of Constraints

The pre- and guard- conditions used in a use case specification capture the conditions under which a given use case scenario is executed, in other terms, they allow to determine the inputs that enable the covering of a given use case scenario. UMTG automatically identifies these conditions by means of NLP and asks the software engineer to reformulate them as OCL constraints. In later steps, UMTG uses a constraint solver to solve the provided OCL constraints and automatically generate test inputs (see Section 3.7). Postconditions instead capture the expected state of the system after the execution of the scenario. UMTG asks the software engineer to reformulate postconditions as OCL constraints as well. Postconditions are later translated into oracles for the test cases.

To minimize manual effort, the NLP application first locates use case conditions and then identifies repeating and negated ones through an NLP transducer. If the use cases both feature the condition and its negation, only the condition is reformulated as an OCL constraint and its negated version in OCL is automatically derived. Table 3.2 provides some of the use case conditions in Table 2.2 with their corresponding OCL constraints based on the domain model.

3.6 Generation of the Use Case Test Model

A *Use Case Test Model* enables the mapping of use case steps with test case steps. It makes the implicit control flow in a use case specification explicit. Figure 3.6 gives the metamodel for the *Use Case Test Model*.

The *UseCaseStart* represents the beginning of a use case and is thus associated to a precondition and to the first step in the use case. *Steps* can be of two different types: *Sequence* and *Condition*. A *Sequence* has only a single successor, while a *Condition* has two possible successors. A *Condition* has a reference to a constraint and it is associated with the steps taken according to the truth value of the constraint.

A *Constraint* has a description informally written in the use case (the first column in Table 3.2) and a corresponding OCL constraint referring to the domain model (the second column in Table 3.2). *ComplexConstraints* and *SimpleConstraints* are introduced to reduce the number of manually written OCL constraints. *ComplexConstraints* simply concatenate *SimpleConstraints* using the operators *OR* and *AND*.

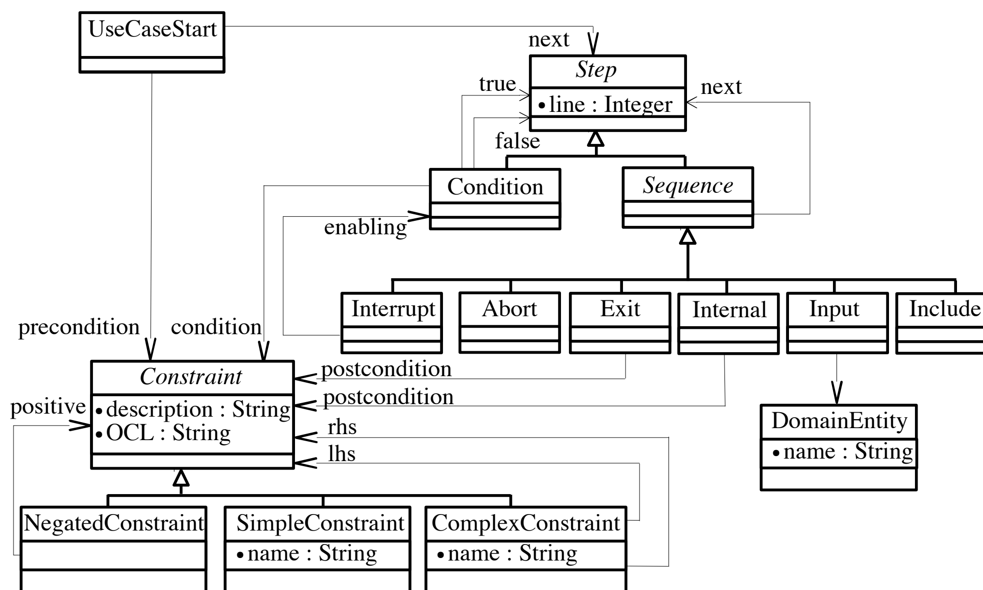


Figure 3.6. Metamodel for the Use Case Testing Model

An *Input* Step indicates that a test case should invoke an input operation during test case execution. *Input* steps reference a *DomainEntity* that represents the data passed as input to the system. An *Interrupt* indicates the presence of inputs that enable the evaluation of a condition of a Global or Bounded Alternative Flow that may disrupt the execution of the basic flow. For this reason, it is linked to the corresponding Condition step. This is further detailed in Section 3.7.

An *Internal* Step indicates that the system alters its internal state. In order to specify the effects of an internal step on the system state, the *Internal* Step is associated to an OCL constraint specified by the software engineer.

An *Exit* Step is the last step of a use case flow and contains a reference to the corresponding postcondition. An *Exit* Step for the Basic Flow does not point to any further step, while the *Exit* Step of an Alternative Flow points to the step indicated by the keyword *Resume* in the use case. *Abort* Steps instead terminate an anomalous execution flow and do not point to any further step.

Figure 3.7 shows the Use Case Test Model generated from the use case specification in Table 2.2 (to improve readability, the figure shows only the objects that model the execution flow).

The Use Case Test Model for a use case is generated by processing the use case specification annotated by the NLP application. UMTG generates a Use Case Test Model for each use case specification. The use case document is parsed from the beginning to the end. Each time a textual element tagged as *Input*, *Include*, *Internal*, or *Conditional* is encountered, the technique generates a corresponding Step instance and connects it to the last Step instance created.

Whenever a domain entity is encountered, a corresponding *DomainEntity* object is created (if it does not exists already) and is linked to the Step that refers to that domain entity.

When a Specific Alternative Flow is found, the *Reference* block is used to find the corresponding Condition Step of the Basic Flow and to connect it with the first step of the *Alternative Flow*. The

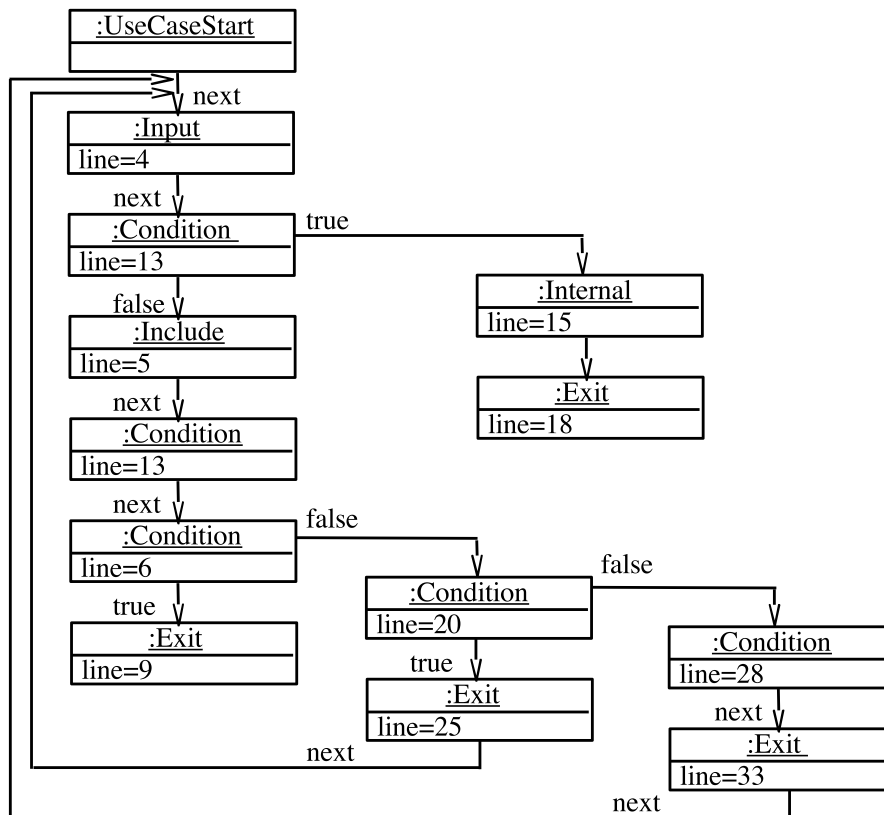


Figure 3.7. The Use Case Testing Model derived from the Use Case in Table 4.1

Condition Step of line 6 in Table 2.2, for example, is connected with the step of line 20 (the first of the Specific Alternative Flow).

Global and Bounded Alternative Flows lead to the generation of multiple Condition Steps. In UMTG, Global and Bounded Alternative Flows begin with a guard condition, i.e. a Condition step. In case of a Global Alternative Flow, the Condition step at the beginning of the Alternative Flow is added before all the steps of the basic flow. For a Bounded Alternative Flow, the Condition step is added before all the steps covered by the Bounded Alternative Flow. In the use case in Table 2.2, the Condition step of line 13 belongs to a Bounded Alternative Flow that is bounded by lines 5 to 6, which results in the Use Case Test Model containing two Step instances *'Condition line 13'* before the steps of lines 5 and 6.

If multiple alternative flows depend on the same condition of the basic flow, UMTG simply connects their guard Conditions in cascade following the order in the use case specification. This is the case of Conditions on lines 20 and 28 of the use case in Table 2.2, which are connected in cascade in Figure 3.7.

3.7 Generation of Scenarios and Test Inputs

Test inputs can be identified from the Use Case Test Model to cover all the possible scenarios of the use case, i.e., all the paths that begin with a *UseCaseStart* step and end with an *Exit* step.

According to our definition, a use case scenario does not include the steps in the Use Case Test

Model that follow an *Exit* step, and thus we can treat the Use Case Test Model as an acyclic graph. For this reason, we implemented a test generation algorithm that covers all the paths of the Use Case Test Model by following a depth-first traversal.

Constraint solvers can be used to solve the path condition that must be satisfied to cover a specific path and thus help identify test inputs. UMTG can work with any constraint solver dedicated to OCL. We have worked with two distinct constraint solvers, one that relies upon search algorithms to efficiently generate test inputs that satisfy the given path condition (please refer to [Ali et al., 2013] for more details), and one based on a model transformation that allows to use the Alloy Analyzer to generate results, i.e. object models, that satisfy the given OCL constraints [Anastasakis et al., 2007].

Require: *step*, the step to inspect
Require: *scenario*, list of steps that constitute the current test case
Require: *pc*, the current path condition
Require: *results*, list of test inputs already generated
Require: *dm*, the domain model
Require: *M*, the Use Case Test Model under test
Ensure: *results*, a list of pairs $< scenario, objectDiagram >$
1: **if** *step is UseCaseStart* **then**
2: $scenario \leftarrow scenario \cup step$
3: $pc \leftarrow pc\ AND\ step.precondition$
4: $GenerateInputs(step.next, scenario, pc, tests, dm, M)$
5: **end if**
6: **if** *step is Input* **then**
7: $scenario \leftarrow scenario \cup step$
8: $GenerateInputs(step.next, scenario, pc, tests, dm, M)$
9: **end if**
10: **if** *step is Condition* **then**
11: *//prepare for visiting the true branch*
12: $scenario_T \leftarrow scenario$ *//create a copy*
13: **if** *step belongs to Global or Bounded* **then**
14: $scenario_T \leftarrow$ **new**$InputInstruction(step.entities)$
15: **end if**
16: $scenario_T \leftarrow scenario_T \cup step$
17: $pc_T \leftarrow pc\ AND\ step.condition$
18: $GenerateInputs(step.true, scenario_T, pc_T, tests, dm, M)$
19: *//prepare for visiting the false branch*
20: $scenario_F \leftarrow scenario$
21: **if** *step belongs to Global or Bounded* **then**
22: $pc_F \leftarrow pc$
23: **else**
24: $scenario_F \leftarrow scenario \cup step$
25: $pc_F \leftarrow pc\ AND\ !\ step.condition$
26: **end if**
27: $GenerateInputs(step.false, scenario_F, pc_F, tests, dm, M)$

28: **end if**
29: **if** *step is Internal* **then**
30: $scenario \leftarrow scenario \cup step$
31: $pc \leftarrow pc\ AND\ step.postcondition$
32: $GenerateInputs(step.next, scenario, pc, tests, dm, M)$
33: **end if**
34: **if** *step is Exit OR Abort* **then**
35: $scenario \leftarrow scenario \cup step$
36: $objectDiagram \leftarrow oclSolver(pc, dm)$
37: $results \leftarrow results \cup < scenario, objectDiagram >$
38: **end if**

Figure 3.8. GenerateInputs: the Test Generation Algorithm adopted in UMTG

The following paragraphs describe GenerateInputs, the algorithm used by UMTG to build and solve the path conditions. Figure 3.8 shows the algorithm.

The algorithm takes six inputs: *step*, the step to inspect, *scenario*, the current test scenario, which is a list of steps covered during the traversal, *pc*, the path condition of the scenario, *results*, a list of results generated, *dm*, the domain model, *UCTM*, the Use Case Test Model for which we intend to generate test cases. The algorithm generates a list of pairs $< scenario, objectDiagram >$. Each pair shows an object diagram that includes the input values enabling the coverage of a specific *scenario*.

UMTG generates the scenario and test inputs of the Use Case Test Model under test by invoking GenerateInputs using the UseCaseStart as the parameter *step*. The lists *scenario*, *pc*, and *results* are initially empty and are populated by GenerateInputs during its execution.

Before executing GenerateInputs, UMTG includes into the Use Case Test Model under test the steps of the included use cases. This is done in a fashion that is similar to the generation of inter-procedural control flow graphs [Harrold et al., 1998], i.e., by connecting the step that precedes the *Include* step with the *UseCaseStart* of the included Use Case Test Model, and by connecting the *Exit* step of the basic flow of the included Use Case Test Model with the step that follows the *Include* step. Additionally, UMTG removes branches that lead to loops in the included Use Case Test Model.

GenerateInputs traverses the Use Case Test Model by following a depth-first traversal and populates the list *scenario* with the steps visited within a path. Figure 3.9 shows three scenarios generated for the running example. The following paragraphs describe the activities performed for the different kinds of steps.

In the case of a *UseCaseStart* step, GenerateInputs adds the pre-condition of the *UseCaseStart* step to the path condition. This is done because it enables the initialisation of the test case (lines 1 to 5 in Figure 3.8).

In the case of a *Condition* step, GenerateInputs visits first the true branch and then the false branch. Visiting the true and the false branch simply consists in recursively invoking GenerateInputs after appropriately updating the path condition. When visiting the true branch, GenerateInputs adds a new *Interrupt* step to the scenario if the *Condition* step belongs to a Global or Bounded Alternative Flow (line 14 in Figure 3.8). This is done because the execution of the true branch of a Bounded or Global Alternative Flow indicates the presence of an interruption. Scenario B in Figure 3.9 shows the *Interrupt* step added before the *Condition* step of line 13. The path condition built to visit the true branch of line 13 is the conjunct of conditions 5 and 3 of Table 3.2.

Lines 19 to 27 in Figure 3.8 handle the visit of the false branch. GenerateInputs adds the Condition step to the scenario only if it does not belong to a Global or Bounded Alternative Flow. This is done because the guard conditions of Global and Bounded Alternative Flows influence the behaviour of a use case only when they are true. If the Condition step does not belong to a Bounded or Global Alternative flow, GenerateInputs adds the negation of the associated OCL constraints to the path condition (line 25). Scenario A in Figure 3.9 corresponds to the basic flow of the use case in Table 2.2. The Figure shows that the scenario does not include the condition of the Bounded Alternative flow starting in line 13 of Table 2.2. An example of the negation of a constraint that does not belong to a Bounded Alternative flow is shown in Scenario C (Figure 3.9). The bold font shows the negated constraint.

An *Internal* step indicates that the system changes its internal state. Since the change may affect

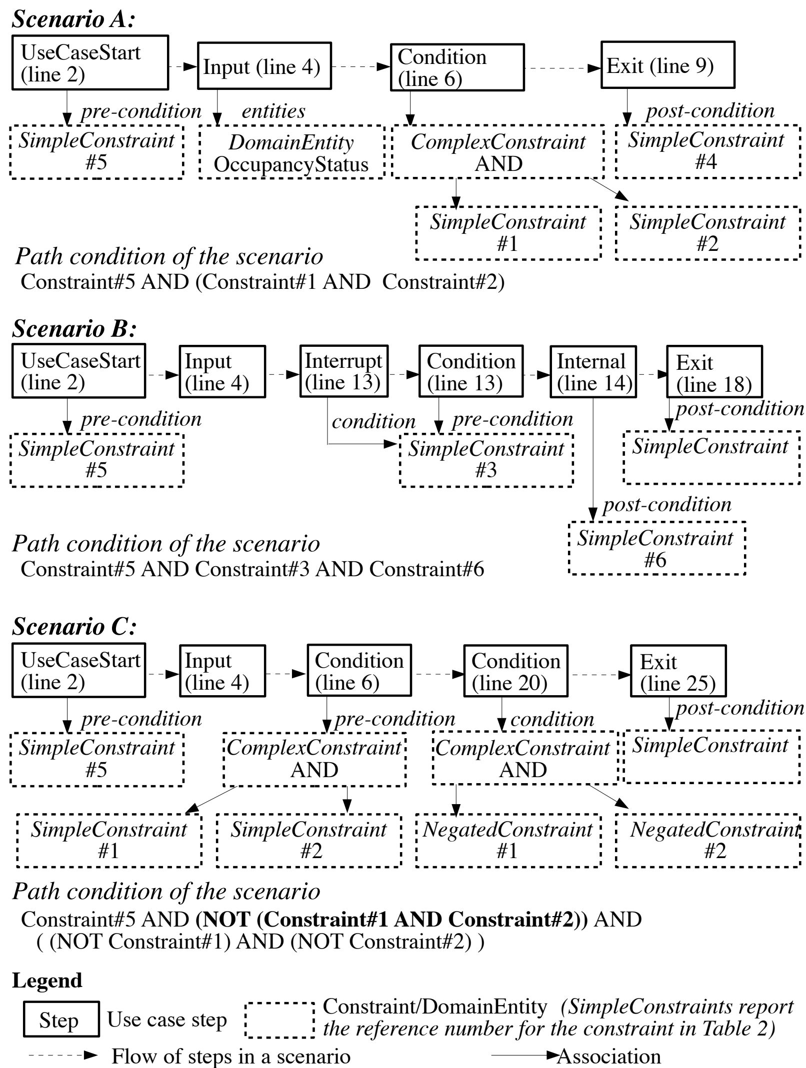


Figure 3.9. Three scenarios built by UMTG for the use case in Table 2.2

the truth value of following Condition steps, GenerateInputs includes in the path condition the constraints associated with the *Internal* steps visited (line 31 in Figure 3.8). An example is given by the path condition of Scenario B in Figure 3.9.

The generation of a scenario terminates when either an *Exit* or an *Abort* step is reached (line 34 in Figure 3.8). GenerateInputs adds the *Exit* or *Abort* step to the *scenario* (line 35) and then invokes the OCL solver to solve the current path condition (line 36). A pair $< scenario, objectDiagram >$ with the current scenario, and the object diagram returned by the OCL solver is added to the list *results* (line 37).

3.8 Generation of Test Cases

UMTG generates test cases by processing all the pairs $< scenario, objectDiagram >$ produced by the algorithm GenerateInputs. UMTG performs three activities to generate test cases: Identify input values, Generate high-level operation descriptions, and Generate calls to driver functions. These

activities are tailored to the specific format described in Section 2.1 but may be changed, following similar principles, to generate test cases in different formats for embedded systems using a different test infrastructure and hardware.

Table 3.3 shows a test case automatically generated from the basic flow of the use case in Table 2.2 (lines 1,3, and 5 are high-level operation descriptions; lines 2,4, and 6 are driver function calls). This test case corresponds to the manually written test case in Table 3.1. Figure 3.10 shows the activities performed by UMTG to automatically generate the test case.

UMTG first processes the object diagram to identify the whole set of input values for the test case (activity 1 in Figure 3.10). To this end, UMTG looks for the attributes in the object diagram that appear in the constraints of the scenario.

UMTG generates high-level operation descriptions by first creating *Input* operations and then *Check* operations. UMTG creates a test line with an *Input* operation for every *UseCaseStart* and *Input* step of the scenario. For each input line in the test case, UMTG selects input values that belong to the domain entities appearing in the corresponding step of the scenario. For instance, in activity 2.2 in Figure 3.10, the inputs *occupancyStatus.occupantClassForAirbagControl = Adult* and *occupancyStatus.occupantClassForSeatBeltReminder = Adult* are selected when processing the *Input* step in line 4 (see Scenario A in Figure 3.9). In fact, the *Input* step in line 4 refers to to the domain entity *OccupancyStatus*.

At the end of the test case, UMTG generates a test line with a *Check* operation and the postcondition of the scenario under test (see activity 2.3 in Figure 3.10).

Table 3.3. A generated test case for *BodySense*

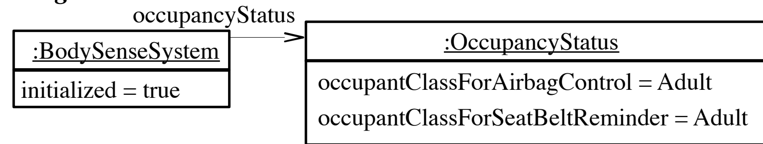
Line No.	Operation	Inputs
1	*Input*	System.initialized = true
2	ResetPower	Time=INIT_TIME
3	*Input*	occupantClassForAirbagControl = Adult occupantClassForSeatBeltReminder = Adult
4	SetBus	Channel=RELAY Capacitance=85
5	*Check*	AirbagControlUnit.allInstances() → forAll(a|a.status = a.bodySense .occupancyStatus.occupantClassForAirbagControl) AND ..
6	ReadAnd CheckBus	D0=OCCUPIED D1=OCCUPIED

Table 3.4. Mapping table for *BodySense* (excerpt)

Matching Patterns (Operation and Inputs)		**Result (Operation and Inputs)**	
Input	.*ClassForAirbagControl = Adult .*SeatBeltReminder = Adult	Set Bus	Channel = RELAY Capacitance = 85
Input	System.initialized = true	Reset Power	Time=INIT_TIME
Check	AirbagControlUnit.allInstances()- >forAll(a|a.status=a.bodySense.occupancy...	ReadAndCheckBus	D0=OCCUPIED D1=OCCUPIED

To generate calls to driver functions, UMTG parses each high-level operation description by using the mapping table provided by the software engineers (activity 3 in Figure 3.10). Table 3.4 shows a

Object diagram:



Test Case Generation:

1. Identify input values

System.initialized = true
System.occupancyStatus.occupantClassForAirbagControl = Adult
System.occupancyStatus.occupantClassForSeatBeltReminder = Adult

2. Generate high-level operation descriptions

2.1 Process the UseCaseStar step of line 2: add the following line to the test case

Input	System.initialized = true

2.2 Process the Input step of line 4: add the following line to the test case

Input	System.occ...tus.occupantClassForAirbagControl = Adult System.occ...tus.occupantClassForSeatBeltReminder = Adult

2.3 Process the Exit step of line 9: add the following line to the test case

Check	AirbagControlUnit.allInstances() - > forAll(ala.status = a.bodySense.occupancyStatus .occupantClassForAirbagControl) AND SeatBeltControlUnit.allInstances() - > forAll(sls.status = s.bodySense.occupancyStatus.occupantClassForSeatBeltReminder)

3. Generate calls to driver functions
Lines 2, 4, and 6 in Table 4 are generated by processing the high-level operation descriptions according to the Mapping Table

Figure 3.10. Activities performed by UMTG to generate the test case in Table 3.3

mapping table for *BodySense*. The mapping table is made of four columns. The first two columns provide operation names and regular expressions that match high-level operation descriptions in the test case. The last two columns provide the driver function calls that should be added after a match of a high-level operation description. For example, line 1 of the test case in Table 3.3 matches the expression *'System.initialized = true'* and thus leads to the generation of line 2 of the test case.

UMTG generates test oracles by translating OCL postconditions into the scripting language of the testing framework. In our current implementation the translation is simply performed by means of the mapping table (see the third row in Table 3.4). As a future work, we plan to integrate more complex strategies to perform the translation.

3.9 Empirical Evaluation

We have conducted an empirical evaluation of UMTG that aims to evaluate its applicability and effectiveness based on an industrial case study, *BodySense*.

The empirical evaluation aims to respond to two research questions:

- *RQ1. Is the modelling and analysis effort required by UMTG acceptable in an industrial context?* UMTG requires use case specifications written according to a specific format (RUCM), plus the specification of OCL constraints that correspond with constraints written in natural language. This research question aims to evaluate whether the required effort is acceptable in

an industrial context.

- *RQ2. Is UMTG effective in generating test cases that cover the requirements captured by use case specifications?* UMTG aims to generate test cases that demonstrate that a software implementation complies with its functional specifications. The goal of this research question is to evaluate whether UMTG allows to generate test cases that cover all the functional requirements of the system.
- *RQ3. Does UMTG scale?* UMTG aims to generate system test cases which might in principle require the generation of complex inputs after solving complex OCL constraints. RQ3 aims to evaluate whether the approach scales to the size of industrial projects.

3.9.1 RQ1. Is the modelling and analysis effort required by UMTG acceptable in an industrial context?

In order to respond to RQ1 we both conducted informal interviews with IEE engineers, and measured the additional effort required by IEE engineers in order to produce the input artefacts required by UMTG. More specifically, we measured the number of scenarios that need to be defined to write six use case specifications of *BodySense* according to RUCM, and we compare this number with the number of scenarios specified by IEE engineers before adopting RUCM. Also we counted the number of OCL constraints that need to be provided by engineers in order to apply UMTG.

Table 3.5. Results obtained with the case study

Use Case ID	**Use Case Flows**		**OCL Constraints**
	IEE	UMTG	
1	6	8	9
2	11	13	7
3	4	8	8
4	7	11	12
5	7	8	5
6	6	6	12

Table 3.5 shows the collected measurements. Column *Use Case Flows-UMTG* shows that the RUCM format lead to use case specifications that include more use case flows than the original specifications written by IEE engineers (Column *Use Case Flows-IEE*), this mostly depends on the fact that RUCM forces engineers to avoid ambiguous requirement steps.

The column *OCL Constraints* shows the number of constraints we specified by using OCL. The table shows that the number of constraints to define for each use case is low, ranging from 5 to 12. Furthermore several constraints are shared by multiple use cases, engineers thus had to define 48 unique constraints, less than the sum of the constraints of each use case (53).

Also, we report that the OCL constraints we had to specify are relatively simple and easy to express. Only 12 constraints are complex constraints that result from the conjunction of simple constraints. Furthermore most of the obtained constraints correspond to equalities (the operator '=' appears 28 times, while the operators '$<$' and '$>$' appear 11 and 8 times respectively).

Finally we report that IEE engineers consider the modelling effort required by UMTG acceptable, and also, at the time of this dissertation writing, they have started adopting the RUCM to elicit the use case specifications for ongoing projects.

3.9.2 RQ2. Is UMTG effective in generating test cases that cover the requirements captured by use case specifications?

To evaluate UMTG effectiveness and compare it with manual testing we measured the amount of scenarios covered by the test cases generated with UMTG, and compared with the scenarios covered by the test cases manually written by IEE engineers, based on domain expertise.

The last two columns of Table 3.6 show the number of use case scenarios covered by both the test cases manually written by IEE software engineers, and by UMTG. The table shows that UMTG covers use case scenarios that are not covered by manually derived use cases. This is mainly due to the fact that test engineers tend not to test all the scenarios that can be derived from bounded or global alternative flows (a same bounded or global alternative flow may lead to multiple paths, i.e. scenarios, in the Use Case Test Model). Doing that type of path analysis manually is indeed difficult for an engineer. Another reason for this discrepancy is that RUCM is a more precise description of use case specifications than standard use cases, thus leading to the identification of more scenarios.

Table 3.6. Results obtained for RQ2

Use Case ID	**Scenarios Covered**	
ID	IEE	UMTG
1	30	37
2	10	13
3	8	8
4	24	28
5	4	8
6	4	6

3.9.3 RQ3. Does UMTG scale?

To respond to *RQ3* we measured the execution time required by UMTG to generate test cases, in addition, to discuss the generalizability of the results we report the number of steps of each use case specification, and the number of arcs and node of the generated testing models.

Most of the computation time during test case generation was spent on constraint solving. As mentioned in Section 3.7 in our experiments we relied upon two different constraint solvers, one based on Alloy, and one dedicated to OCL which based on a search algorithm [Ali et al., 2013].

We noticed that the execution time required to generate test cases highly varies depending on the constraint solver adopted. In the case of the solver dedicated to OCL the automatic generation of a test case, for each use case scenario, approximately took 12 minutes on average, with a maximum

execution time of 56 minutes, and a minimum execution time of one minute. In the case of Alloy the execution time was much lower, 10 seconds per test case on average.

The identification of the constraint solver to be used in our context is out of the scope of this dissertation, however, these numbers clearly indicate that the approach scales since, in the worst case, test case generation can be run overnight and each test case could easily be generated in parallel on a high-performance computing platform.

The data about the size of test cases and testing models are reported in Table 3.7 and shows that the use case specifications considered for our evaluation are non-trivial, including from 25 to 50 steps each. Also the Table shows that the number of nodes and arcs in the UCTM models is not small, which makes us believe that the considerations about the scalability of the approach may generalize to other industrial embedded systems.

Table 3.7. Results obtained for RQ3

Use Cases			**UCTM**
ID	Steps	Flows	Nodes, Arcs
1	50	8	154, 182
2	44	13	46, 55
3	35	8	35, 40
4	59	11	119, 140
5	30	8	32, 38
6	25	6	27, 30

3.9.4 Threats to Validity

Internal threats

Threats to the internal validity of our empirical observations may depend on a potentially wrong implementation of the toolset for generating the test cases. For this reason, to avoid errors depending on a wrong implementation of our toolset we manually inspected both the use case testing models generated by UMTG, the abstract test cases, and the final test cases generated by the technique.

External threats

Threats to the external validity regard the generalisability of our findings. Given that *BodySense* must comply with safety standards we consider the test cases developed by IEE engineers being an example of good manual test cases. Thus, we expect that the test cases provided by IEE engineers can test most of the foreseeable execution scenarios. Also, we stress that *BodySense* is a representative automotive embedded system developed by IEE which have been sold for many years. For all these reasons we assume that our considerations about the effectiveness of UMTG test suites, which improves the coverage of the manual test suite implemented by IEE engineers, might generalize to many other embedded systems.

3.10 Conclusion

In this chapter, we presented an automated approach for the generation of functional system test cases. The approach is driven by use case specifications, augmented with a domain model.

The approach combines NLP and constraint solving to extract implicit behavioural information from use case specifications, and help generate test scenarios and data. In addition, to enable test data generation the approach requires that guard, pre, and post conditions of RUCM are refined by means of OCL expressions based on the domain model.

The industrial case study shows that our automated approach works with industrial use case specifications for a representative automotive embedded system. The time required for test case generation enables the entire process to run overnight, and furthermore, it could easily be parallelised if required. Our experience also suggests that the modelling requirements of our approach, in terms of use case specifications and domain modelling, is feasible in an industrial context.

Chapter 4

System Testing of Timing Requirements based on Use Cases and Timed Automata

This chapter presents *Test Generation combining Timed Automata and Use Case Specifications*, TAUC, a technique that automatically generates executable test cases targeting software timeliness.

TAUC automatically builds the detailed models necessary to generate executable test cases. This is achieved by combining the information appearing in use case specifications and models of the timing requirements of the system (timed automata). It thus prevents software engineers from designing very detailed timed automata when use case specifications are available.

In addition, TAUC automatically builds test suites that exercise the functionality regulated by timing requirements by optimising diversity among test inputs, thus increasing the probability of identifying problems dependent on specific input sequences.

To automate testing, TAUC needs to determine what functional inputs are needed to reach certain system states, and for this reason, TAUC must identify the dependencies between functional scenarios and timed automata. More specifically, TAUC determines which functional scenarios bring the system into a specific state, and which functional scenarios can take place when the system is in a given state. TAUC relies upon UMTG to identify the test inputs that exercise functional scenarios and automatically processes test inputs and timed automata to identify their dependencies.

TAUC automatically models these dependencies by augmenting the set of user-provided timed automata capturing timing requirements. This helps contain the engineers' modelling effort and enables the use of UPPAAL [Bengtsson et al., 1995], a model checker for symbolic reachability analysis of timed automata. TAUC relies upon UPPAAL to generate test cases that include both inputs derived from use case specifications and timing constraint on the inputs, e.g. the delay between two inputs, derived from timed automata.

Test generation with UPPAAL guarantees edge coverage, i.e. functional coverage, but does not aim at increasing the chance to identify the violation of timing requirements. For this reason, TAUC includes a meta-heuristic search algorithm that iteratively modifies the test cases generated by UPPAAL to maximize test suite diversity within a certain test budget. Test suite diversity is maximized by deriving test cases that provide diverse sequences of inputs to the system, thus increasing the

probability of identifying violations of timing requirements that depend on specific input sequences.

The contributions of this chapter are summarized as follows:

- A model-based, automated strategy to test timeliness requirements at the system level, in the context of use case driven development.
- The definition of timeliness test models, i.e., models based on the timed automata formalism that enables the automated generation of executable test cases.
- A strategy for generating timeliness test models from functional and timing requirements provided as use case specifications and timed automata. This strategy allows to minimize modeling overhead and automates test generation targeting timeliness requirements.
- An empirical evaluation based on *BodySense* that shows the effectiveness of the approach.

The chapter proceeds as follows. In Section 4.1 we discuss the limitations of existing approaches for generating test cases for software timeliness that motivate the development of TAUC. In Section 4.2 we overview the benefits given by TAUC. In Section 4.3 we provides an overview of the steps taken by TAUC to automate test generation. In Sections 4.4 to 4.7, we provide details about TAUC steps. In Section 4.8 we discuss the empirical results obtained with *BodySense*. In Section 4.9 we provide final remarks.

4.1 Limitations of Test Generation Based on Timed Automata

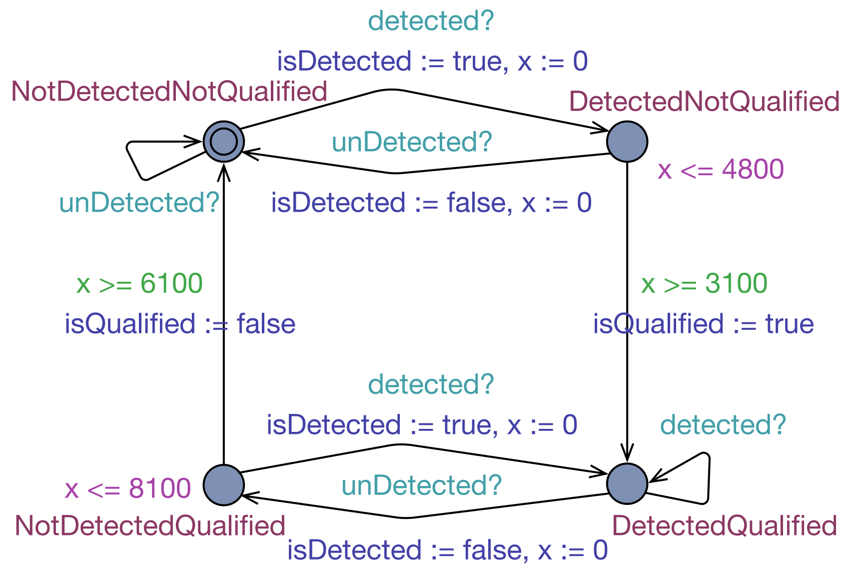


Figure 4.1. Automaton that captures how TemperatureErrors are qualified and dequalified in *BodySense*. This is the same automaton shown in Figure 2.2. Edges are labelled by the triple guard (green), action (light blue), and update (blue).

Software engineers may use the timed automaton of Figure 4.1 to automatically generate timeliness abstract test cases by using an automated technique [Hessel et al., 2004]. The first three operations performed by an automatically generated test case might be: (1) generate the event *detected* (which is supposed to bring the system into the state *DetectedNotQualified* from the initial state *Not-*

DetectedNotQualified), (2) wait for 4801 milliseconds (to be sure that the edge has been fired), (3) check that the system is in the state *DetectedQualified* and otherwise fail.

To transform this abstract test case into an executable test case, the software engineer needs to determine how to generate the event *detected*. This activity is not trivial because one has to carefully read the software specifications to determine the conditions under which a temperature error is detected. For a complex system, finding all the temperature requirements could be particularly expensive. For example, the engineer might need to read all the specification documents to be sure that the temperature range does not vary according to working conditions (e.g. in the presence of other system errors).

4.2 Automated Testing with TAUC

TAUC assumes that software engineers have produced use case specifications written in the RUCM format with OCL constraints defined by following the UMTG approach. Also TAUC requires timed automata that capture timing requirements as shown in Chapter 2.

Table 4.1. *BodySense* Use Cases

0	**Use Case: Identify the Occupancy Status of a Seat**
1	**Precondition**
2	The system has been initialized
3	**1.1 Basic Flow**
4	1.The system REQUESTS capacity FROM the seat sensor.
5	2.INCLUDE USE CASE Self diagnosis.
6	3.INCLUDE USE CASE Classify occupancy status.
7	4.The system VALIDATES THAT no error is detected and no error is qualified.
8	5.The system VALIDATES THAT the occupant class is valid.
9	6.The system SENDS the occupant class TO AirbagControlUnit.
10	Postcondition: The occupant class has been sent to AirbagControlUnit.
11	**1.2 Specific Alternative Flow**
12	RFS 4
13	1. The system sends the error class to AirbagControlUnit.
14	2. ABORT.
15	Postcondition: Classification filters have been reset.
...	...
30	**Use Case: Self Diagnosis**
31	**2.1 Basic Flow**
32	1.The system REQUESTS temperature FROM the temperature sensor.
33	2.The system VALIDATES THAT the temperature is valid.
...	...
37	**2.2 Specific Alternative Flow**
38	RFS 2
39	1. The system sets the TemperatureError as detected.
40	2. RESUME 3.

Table 4.1 shows two use cases of *BodySense*, *Identify the Occupancy Status of a Seat* (Line 0) and *Self Diagnosis* (Line 30). The former describes the main functionality of *BodySense*, while the latter deals with the identification of runtime errors, e.g. it detects the presence of a *TemperatureError* if the temperature measured by the sensor is out of range (Line 39). This is a refined version of the use case presented in Table 2.2 that contains only the steps required to provide a clear understanding of TAUC.

To enable test case generation, UMTG requires that software engineers specify constraints expressed in the Object Constraint Language (OCL [OMG, 2004]), based on the domain model. These

constraints precisely specify conditional statements, effects of internal steps, and post-conditions of use case flows.

The constraint *TemperatureSensor: self.temperature.value > 0 and self.temperature.value < 40* for example, is used to further specify the conditional statement in Line 33 (*the temperature is valid*) and shows that the temperature is valid when its value lies between 0 and 40.

The test generation process implemented by TAUC guarantees that all the edges of the timing requirements automata are covered at least once and that, furthermore, test suites contain test inputs that are combined in ways to maximize their diversity.

For example, TAUC may generate a timeliness test case for *BodySense* that begins in the state *NotDetectedNotQualified* (initial state of the system), simulates the sending of the value 76 from the temperature sensor (in this way the edge is fired and the new active location is *DetectedNotQualified*), waits for 4801 milliseconds, and checks that the system is in the state *DetectedQualified*. In contrast to the test case generated with a traditional approach (Section 4.1), the test case generated by TAUC does not require that the software engineer manually determines that an input temperature with a value above 40 is needed to reach location *DetectedNotQualified*. The concrete test input to be used is automatically generated by TAUC.

To identify the concrete test inputs to be used in a test case, TAUC relies upon the identification of dependencies between functional scenarios and timed automata. These dependencies are used to determine which functional scenarios need to be exercised to bring the system to a specific state. In the case of *BodySense*, TAUC automatically detects that there is a dependency between the event *detected* on the automata in Figure 4.1 and the use case step of Line 39, *The system sets the TemperatureError as detected*. This dependency enables TAUC to determine that the inputs that exercise the use case scenario that covers Line 39, i.e. an object diagram that assigns the value 76 to the temperature sensor, is necessary to bring the system into the state *DetectedNotQualified*.

TAUC also generates test cases that highly differ from one another, to maximize the diversity of the test suite. A test case, different from the one presented above, is generated by TAUC to send an interrupt, followed by a message, while the system is in the location *DetectedNotQualified*. Such test cases check the effect of different inputs on timing requirements. Details are provided in the coming sections.

4.3 Overview of TAUC

When use case specifications are used to specify the software functional behaviour, TAUC spares software engineers from the burden of manually determining the functional inputs required for testing timing requirements. TAUC works in six steps, shown in Figure 4.2.

In Step 1, *analysis and design*, software engineers produce RUCM use case specifications that capture the functional requirements of the system, and timed automata that capture both the timing requirements of the system and the timing properties of the environment.

In Step 2, *identification of functional scenarios*, the technique relies upon UMTG to automatically

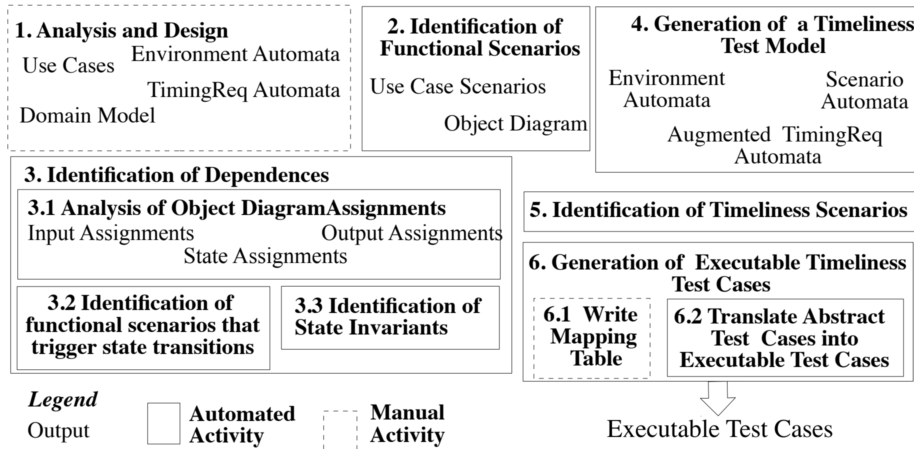


Figure 4.2. Steps of TAUC.

identify functional scenarios, and the inputs required to trigger each scenario.

Step 3 concerns the *identification of the dependencies between functional scenarios and timed automata*. We identify two types of dependencies: (1) the outputs produced by the system during the execution of specific functional scenarios may correspond to events that trigger state transitions, (2) specific functional scenarios might be executed only in certain system states, i.e. only when specific state invariants hold. TAUC automatically identifies these dependencies.

In Step 4, TAUC generates a Timeliness Test Model, i.e. a collection of models that include environment automata provided by software engineers in Step 1, automata generated by TAUC to model functional scenarios (scenario automata), and automata generated by TAUC by extending timing requirements automata. Timeliness Test Models capture the dependencies between functional scenarios and timing requirements automata, and thus enable the adoption of UPPAAL for the generation of executable test cases targeting timeliness.

In Step 5, TAUC derives *timeliness scenarios*. A timeliness scenario is a sequence of delays, edges and locations of the Timeliness Test Model, that specifies a valid execution. A single timeliness scenario generally covers multiple functional scenarios. TAUC generates an initial suite of *timeliness scenarios* with UPPAAL. This initial test suite is then extended by TAUC using a meta-heuristic search strategy that maximizes diversity among the *timeliness scenarios* in the test suite.

In Step 6, TAUC generates *executable test cases* from the timeliness scenarios derived in Step 5. This activity is performed by means of a mapping table provided by software engineers.

The next sections describe in details the different steps of TAUC with the exception of Step 1 and Step 2, which respectively coincide with the modelling of timing requirements as indicated in Section 2.3, and the execution of *UMTG*, already described in Chapter 3.

4.4 Identification of dependencies

Dependencies between timed automata and functional scenarios are of two kinds: (1) outputs produced by functional scenarios may fire state transitions, (2) certain functional scenarios can be exe-

cuted if and only if the system has reached specific states, i.e. specific state invariants are true.

To identify these dependencies, TAUC works in three steps: First, TAUC looks for scenario outputs and assignments to state variables by analyzing the object diagram associated by *UMTG* to each specific scenario. Then it compares scenario outputs and update operations in the automata to identify the edges fired by each scenario. Finally, it relies upon state assignments to identify the state invariants of each functional scenario.

4.4.1 Analysis of attributes in object diagrams

To identify the dependencies between scenarios and timed automata, TAUC first classifies each value assigned to the attributes of the object diagrams generated by UMTG. This is done to distinguish between the inputs required to exercise a functional scenario, the state invariants that must hold when a functional scenario is executed, and the output generated during the execution of the scenario.

In the following, after recalling the characteristics of the output generated by UMTG, we provide details about the process adopted by TAUC to classify attributes in object diagrams.

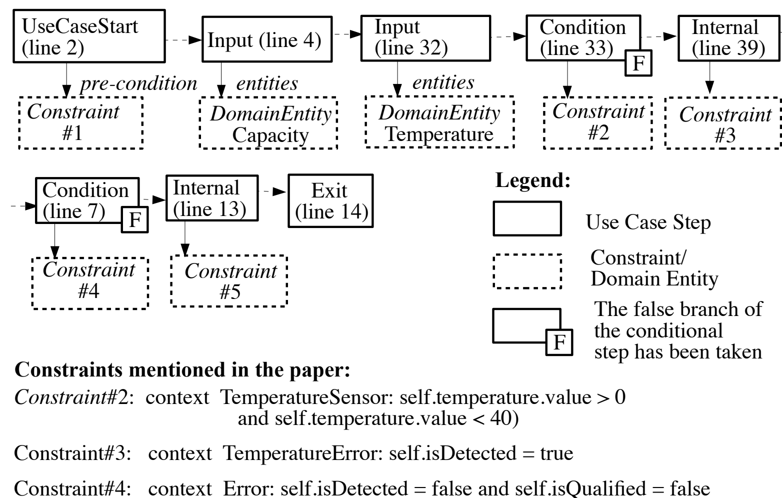


Figure 4.3. Model of a functional scenario that covers the Alternative Flow 2.2 in Table 4.1.

Identification of Test Inputs with UMTG Recall that for each scenario under test, *UMTG* builds a path condition and uses a constraint solver dedicated to OCL to identify an instance of the domain model, i.e. an object diagram, for which the path condition evaluates to true. *UMTG* identifies a set of *functional scenarios*, i.e. sequences of steps in a use case specification, that ensure that the basic flow and each alternative flow of the use case specifications are covered at least once.

Figure 4.3 shows the functional scenario that covers line 39 of *BodySense* use case specifications. Each scenario is a sequence of steps in the use case specification. *UMTG* associates to each step the following information: Input steps are linked to the domain entities received as input by the system. Conditional steps are linked to the OCL constraints that further specify them, e.g. the step of Line 33, which detects the presence of a temperature value out of range, is linked to constraint #2 (see Figure 4.3). Internal steps are linked to postconditions that characterize the state resulting from their execution. The internal step of Line 39, for example, is linked to constraint #3, which indicates that the system has set *TemperatureError* as being detected (see Figure 4.3).

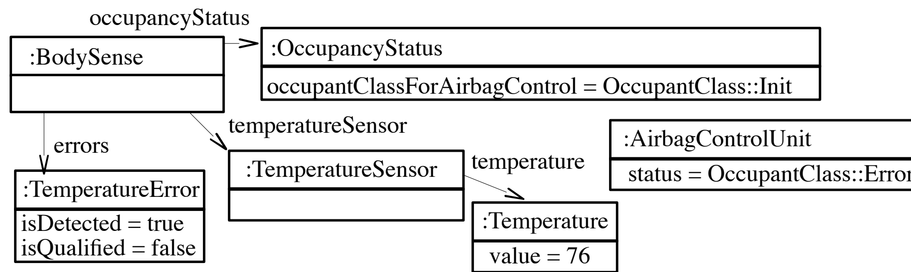


Figure 4.4. Object diagram generated by *UMTG* to satisfy the path condition that covers the scenario in Figure 4.3.

Figure 4.4, shows an object diagram that satisfies the path condition required to exercise the scenario in Figure 4.3.

Analysis of Attributes TAUC classifies the assignments in the object diagrams as *input*, *output*, and *state*.

Output assignments set values to satisfy the post-conditions of internal steps. *Input assignments* and *state assignments* instead set values that satisfy the constraints associated with the conditional steps of a scenario. Assignments to attributes of domain entities appearing in input steps are considered *input assignments*. Assignments to attributes of domain entities not associated with input steps are *state assignments* (attributes that are not input parameters must be state variables).

Temperature.value:=76	**InputAssignment**: regards an attribute of class Temperature, linked to the input step of line 32.
TemperatureError.isDetected:=true	**OutputAssignment**: generated to satisfy constraint#3, which is linked to an internal step.
TemperatureError.isQualified:=false	**StateAssignment**: generated to satisfy constraint #4. TemperatureError is not linked to any input step.

Figure 4.5. Assignments required to exercise the scenario in Figure 4.3.

Figure 4.5 shows a classification of the assignments appearing in the object diagram of Figure 4.4. The assignment *Temperature.value := 76* in Figure 4.5 is an *input assignment*. Indeed, the attribute *value*, which belongs to the entity *Temperature*, is referred to in the input step of Line 32 (Figure 4.3). The assignment *TemperatureError.isDetected := true* is an *output assignment* since it satisfies the *constraint #3*, which is associated with the InternalStep of Line 39 (Figure 4.3). The assignment *TemperatureError.isQualified := false* is a *state assignment* since it satisfies the negation of *constraint #4*, which is associated with the conditional step of Line 7 (Figure 4.3), and TemperatureError is not an input, thus implying *TemperatureError.isQualified* is a state variable.

4.4.2 Identification of functional scenarios that trigger state transitions

In Chapter 2.3 we have anticipated that we expect that software engineers adopt *scenario events* to reduce modelling effort. To further reduce engineers effort, TAUC automatically detects the functional scenarios that dispatch each *scenario event*.

By relying upon *scenario events*, software engineers hide details about the complex relationships between system inputs and the firing of an edge. For example in the automaton of Figure 4.1, the edge that connects states *NotDetectedNotQualified* and *DetectedNotQualified* is fired upon the reception of the scenario event *detected*. The model does not specify the input constraints under which the edge is fired, but shows the effect of the execution of this edge on the system state, i.e. the assignment of the value *true* to variable *isDetected*.

Since *scenario events* trigger edges that update the system state, to automatically detect the scenarios that dispatch scenario events, TAUC assumes that a functional scenario triggers a scenario event if it brings the system into the same state, i.e. if it generates a set of *output assignments* that is a superset of the assignments in the update operations associated with the edge that consumes the scenario event.

The functional scenario in Figure 4.3 leads to the output assignment *TemperatureError.isDetected:=true* (see Figure 4.5), which appears also in the updates of the edge that connects locations *NotDetected-NotQualified* and *DetectedNotQualified* (see the timed automata of Figure 4.1). For this reason, TAUC determines that the scenario in Figure 4.3 is the one that dispatches the event *detected* that triggers the edge between the two above-mentioned locations.

4.4.3 Identification of state invariants

Functional scenarios are enabled only when specific state invariants hold. For example, in *BodySense*, a scenario that performs self diagnosis in the absence of error conditions cannot be exercised if the system already detected errors in previous executions.

State invariants are implicitly specified in use case specifications by means of constraints that capture properties of state variables. TAUC makes the state invariants associated with each functional scenario explicit by identifying the constraints that determine the value of each state assignment. State assignments aim to satisfy conditions that must hold to execute a specific functional scenario and these conditions are thus state invariants.

In the running example the state assignment *TemperatureError.isQualified:=false* results from Constraint #4 in Figure 4.3. This constraint is thus a state invariant that must hold to execute the functional scenario.

In the presence of constraints that relate both input and state variables, TAUC extracts the subexpression concerning state variables only. In the presence of multiple state assignments, TAUC builds a state invariant that joins all the constraints that determine the value of the different state variables.

4.5 Generation of the Timeliness Test Model

We use the term Timeliness Test Model to indicate a network of communicating timed automata that enable the generation of timeliness test cases.

Timeliness Test Models include the environment automata, scenario automata that capture the behaviour of functional scenarios, and augmented timing requirements automata. Scenario automata

and augmented timing requirements automata are automatically generated by TAUC. Augmented timing requirements automata include guard conditions and events that capture the dependencies between timing requirements automata and functional scenarios. The Timeliness Test Models capture all the information required for test generation in the form of timed automata, thus enabling TAUC to rely upon UPPAAL for the automatic generation of test cases.

Scenario automata

For each functional scenario, TAUC automatically generates a scenario automaton. Each scenario automaton contains two edges.

The first edge of the scenario automaton captures dependencies by means of a guard condition for the state invariant of the scenario, and an update for each output assignment. This means that the edge can be fired, i.e. the scenario is executed, only if the state invariant holds. The firing of the edge updates the state of the system as specified by the output assignments. In other words, the scenario automaton shows when a scenario executes and how it affects the state of the system. Given that state invariants identified in the previous step are expressed as OCL constraints, TAUC translates these constraints in a format compatible with the UPPAAL syntax. According to our experience, the set of primitive types supported by UPPAAL (boolean, integer, and double) is rich enough to not limit the kinds of constraint that can be translated in practice.

Figure 4.6 shows an example of generated automaton. The guard condition on the first edge indicates that TemperatureError should not have been already qualified at the beginning of the scenario (as shown by the guard condition *TemperatureError.isQualified == false*). The update operations on the edge capture the scenario output by setting the TemperatureError as detected (see the assignment *TemperatureError.isDetected := true*).

The second edge of the scenario automaton is used to synchronize scenarios and augmented timing requirements automata. This edge generates the event *scenario*, which is consumed by the augmented timing requirement automata, and indicates that the scenario has been executed to completion. This edge also presents a guard condition that captures the scenario frequency (provided by the software engineer). In the case of *BodySense*, each scenario is executed every 1700 milliseconds (See Figure 4.6).

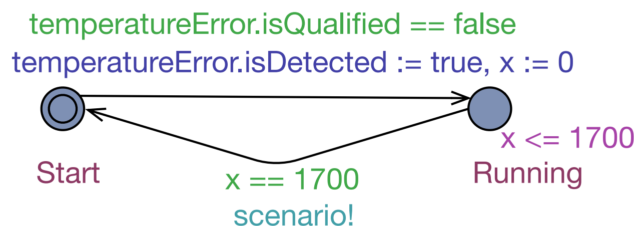


Figure 4.6. Automaton for the Scenario in Figure 4.3

Augmented timing requirements automata

The next step is to modify and augment timing requirements automata. TAUC proceeds as follows. For each scenario event mapped to a scenario output, TAUC replaces the event name in the timing

requirements automata with the synchronization event *scenario*?, which indicates that the event is dispatched only if a scenario had been executed to completion.

Furthermore, TAUC adds a guard condition that checks if the output of the scenario just executed corresponds with the output assignment mapped to the scenario event. This way an edge with a scenario event is fired only if the scenario that generates the corresponding output assignment has been executed.

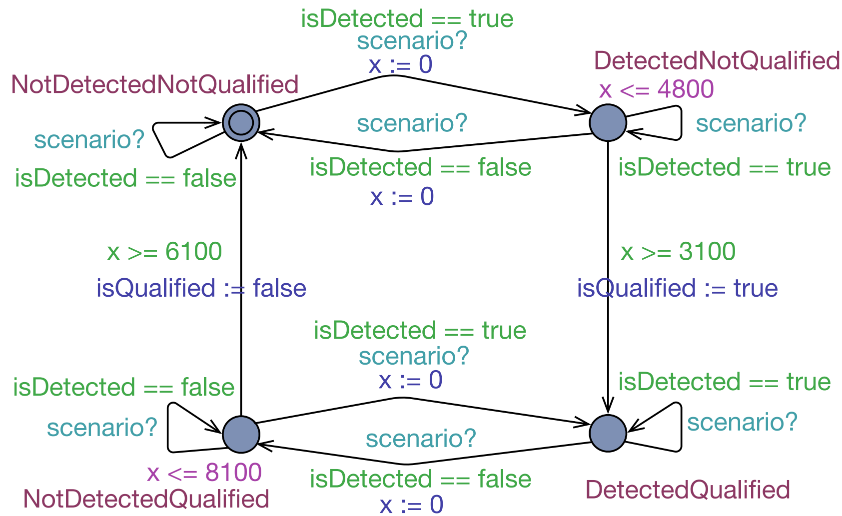


Figure 4.7. Augmented Timing Requirements Automaton derived from the automaton in Figure 4.1.

Figure 4.7 shows, for example, that the edge from location *NotDetectedNotQualified* to location *DetectedNotQualified* is fired only if a functional scenario has been executed and *temperatureError* has been set to detected. The latter is captured through a guard condition which distinguishes different scenarios.

4.6 Identification of Timeliness Scenarios

TAUC automatically generates the timeliness test suite for the system, which consists of a set of timeliness scenarios. A timeliness scenario is a sequence of delays, edges and locations of the Timeliness Test Model, that specifies a valid execution.

A minimal test adequacy criterion to test software timeliness is to exercise all the timing constraints at least once. This is achievable by relying upon a test suite that covers all the locations with invariants including clock variables and all the edges with guard conditions including clock variables. In practice, this condition can be easily satisfied by a test suite that achieves edge coverage. However, in addition to this, we should consider that some failures may only be triggered by specific sequences of test inputs that might not be generated by achieving edge coverage.

In our context, we thus need to generate timeliness scenarios in such a way that, within a test budget, we achieve edge coverage and maximize our chances to find a sequence of inputs that triggers a failure. Moreover, these two goals should be achieved without relying upon the execution of test cases during test generation. In fact in many embedded systems, the test budget, i.e. the number of

test cases that can be executed, is limited because test execution is particularly expensive, e.g. the test execution environment and hardware must be manually set up. Given our objectives and constraints, we present below a test strategy based on maximizing diversity among timeliness scenarios.

Our strategy to increase the chances to trigger a failure, is to generate a test suite that: 1) executes as many diverse paths as possible that include the same edges relevant to timeliness, 2) executes paths with a maximum diversity of inputs, interrupts, and messages. However, in the presence of multiple timed automata, the search space for the identification of execution sequences is large and, for a given test budget, cannot be exhaustively explored to identify an optimal diverse subset of timeliness scenarios. For this reason, the test generation process implemented by TAUC relies upon a metaheuristic search algorithm that maximizes the differences among the timeliness scenarios in the test suite.

The algorithm performs four activities described in the following paragraphs: *Generation of Scenarios That Cover All Edges*, *Generation of New Scenarios*, *Computation of Similarity Score*, *Identification of Scenarios that Maximize Diversity*.

Generation of Scenarios That Cover All Edges

To achieve edge coverage, TAUC relies upon the approach proposed in [Hessel et al., 2004]. It builds a reachability formula that checks for the existence of a trace for which all the edges are covered. TAUC then relies upon UPPAAL to identify the shortest timeliness scenario (called trace in UPPAAL) that satisfies the given formula.

Some of the events generated by UPPAAL automata correspond to parametric system inputs. In the case of *BodySense* this happens for messages of type *CarInfo*, which have two associated parameters, *velocity* and *beltStatus*. Parameter values are not generated by UPPAAL, but are required to properly execute the system. For this reason, TAUC processes the timeliness scenarios generated by UPPAAL and automatically selects the parameter values to be associated to parametric inputs. TAUC simply assigns to each parameter a valid, randomly selected value. In the case of *BodySense*, we deal with both enumerations and numeric values. Enumeration values are selected from the values indicated in the domain model, while numeric values are randomly selected within a range specified by the software engineer.

Generation of New Scenarios New scenarios are then generated by mutating an existing scenario. To this end, TAUC selects an existing timeliness scenario and a mutation point, i.e. a position in the scenario, and then generates a new scenario by copying the events in the scenario up to the mutation point and then by filling the rest of the new scenario with new events. These new events are identified by employing the simulation functionality provided by UPPAAL, which, given the current state of the automata, returns a list of enabled edges. New events are randomly selected till the same length of the initial timeliness scenario is reached. Values of parametric events are generated as well, by following the same procedure indicated previously.

Computation of Similarity Score The diversity among the timeliness scenarios in a test suite is maximized when the average pairwise similarity between all pairs of timeliness scenarios in a test

Timeliness Scenario 1	**Timeliness Scenario 2**	**Similarity Score**
edge(ScenarioAutomata1.Start -> ScenarioAutomata1.Running)	edge(ScenarioAutomata2.Start -> ScenarioAutomata2.Running)	-2
scenario	scenario	+3
ControlRequest(ClearData)	-	-1
edge(TemperatureError.NotDetectedNotQualified -> TemperatureError.DetectedNotQualified)	edge(TemperatureError.NotDetectedNotQualified -> TemperatureError.DetectedNotQualified)	+3
CarInfo(Driving, UnBuckled)	CarInfo(Driving, Buckled)	+1.5
clock x in [3100,4800]	clock x in [3100,4800]	+3
edge(TemperatureError.DetectedNotQualified -> TemperatureError.DetectedQualified)	edge(TemperatureError.DetectedNotQualified -> TemperatureError.DetectedQualified)	+3

Figure 4.8. Alignment of two Timeliness Scenarios

suite is minimized [Hemmati et al., 2013].

To compute the similarity score, TAUC takes into account differences in both event names and event parameter values. TAUC relies upon the Levenshtein string alignment algorithm to identify matching events [Levenshtein, 1966]. Similarly to [Hemmati et al., 2013], TAUC assigns a score of +3 to matching events, -2 to mismatching events, and -1 to gaps. In addition to this, in the presence of mismatching parameter values, TAUC decreases the score by a value between -1 (gap) and -2 (mismatch). More precisely, TAUC decreases the score by 1 plus the fraction of mismatching parameter values. Figure 4.8 shows the similarity score calculated for the alignment of two timeliness scenarios of *BodySense*.

Identification of Scenarios that Maximize Diversity

To build the timeliness test suite, TAUC first augments the UPPAAL test suite with newly generated timeliness scenarios until the test suite reaches the desired size (budget). Such new scenarios are then iteratively updated to maximize diversity.

TAUC replaces a scenario already in the test suite with a new scenario only if the new scenario augments the diversity of the test suite. More precisely, every time a new scenario is generated, TAUC identifies the scenario that should be replaced to obtain a test suite with the lowest average pairwise similarity. Test generation terminates after a given number of iterations provided by the user.

4.7 Generation of Executable Test Cases

For each *timeliness scenario*, TAUC generates a corresponding *abstract test case*.

The abstract test cases generated by TAUC have the same structure of the abstract test cases generated by UMTG, with the only difference that abstract test cases generated by TAUC include not only input operations and oracles, but also delays. Figure 4.9-a shows an abstract test case generated from a timeliness scenario of *BodySense*. Input operations are followed by the input parameters to

Scenario Steps	Abstract Test Case		Executable Test Case	
1	Input	System.initialized = true	Reset	Time = INIT TIME
	Input	TemperatureSensor.value = 76	SetBus	Pin = TEMPERATURE Value = 76
	Input	...ClassForAirbagControl = Adult	SetPin	Channel = RELAY Capacitance = 85
3	Check	TemperatureError.isDetected = true	CheckBus	TEMPERATURE_ERROR = 01h
4	Input	CarInfo(Driving,Buckled)	SndMsg	MsgID = 3Ah D1 = 1 D2 = 0
5	Wait	4800	Wait	Value = 4800
6	Check	TemperatureError.isQualified = true	CheckBus	TEMPERATURE_ERROR = 81h

Figure 4.9. (a) Abstract Test Case generated from Timeliness Scenario 2, and (b) Executable Timeliness Test Case.

be set, delays are identified by the keyword *wait* followed by the time to wait for, while oracles (identified by the keyword *check* in Figure 4.9-a) consist of a list of expressions that check the values of observable system variables.

To generate test inputs TAUC focusses only on the edges that belong to environment automata and scenario automata. For each edge of environment automata that generates an event, TAUC identifies a corresponding test input (event names correspond to entities of the domain model, see *CarInfo* in Figure 4.9). Every time TAUC encounters the first edge of a scenario automata, it adds to the abstract test case the *input assignments* associated with the scenario to be executed (see Step 1 in Figure 4.9-a). All the other edges of environment and scenario automata are ignored because they correspond to synchronization operations between automata.

To generate delays TAUC takes into account the fact that timing properties are typically characterized by uncertainty expressed in terms of ranges in which edges are expected to be taken. TAUC focusses on the maximum amount of time in the range and introduces a delay using a *wait operation* blocking for the maximum amount of time in the range.

To generate oracles TAUC should ideally add, for each edge, an operation that checks if the expected target location has been reached. However, given that many embedded systems, such as BodySense, do not make their current internal state fully observable, TAUC can be configured to derive an oracle that checks for the values assigned to observable system variables. Figure 4.9-a shows an oracle derived from the edge that connects location *DetectedNotQualified* with location *DetectedQualified* (Step 6 of Scenario 2).

Abstract test cases are then translated into executable test cases by means of mapping tables as described in [Wang et al., 2015]. Mapping tables contain regular expressions that match the inputs in the abstract test cases and allow to replace abstract test inputs with concrete test inputs. Table 4.2 shows a portion of the mapping table used to transform the abstract test case in Figure 4.9-a into the executable test case of Figure 4.9-b.

Table 4.2. Mapping Table for *BodySense*

Pattern to match (Operation and Inputs)		Result (Operation and Inputs)	
Input	TemperatureSensor.value = (*)	SetBus	Pin =TEMPERATURE Value = $1
Check	TemperatureError.isDetected=true	CheckBus	TEMPERATURE_ERROR=01h

4.8 Empirical evaluation

We performed an empirical evaluation of TAUC that aims to respond to the following research questions:

- *RQ1. Is TAUC effective in generating test cases able to detect faults that affect software timeliness?* This question aims to evaluate the effectiveness of TAUC in terms of its capability of identifying faults that affect software timeliness.
- *RQ2. Is the modelling and analysis effort required by TAUC acceptable in an industrial context?* TAUC requires timed automata, use case specifications, and mapping tables. This research question aims to determine whether the cost associated with the production of these artefacts can be justified in an industrial context.

4.8.1 RQ1. Is TAUC effective in generating test cases able to detect faults that affect software timeliness?

To respond to *RQ1* we compared the fault coverage of the *BodySense* test suite manually written by software engineers familiar with the system and domain, with the fault coverage of 10 test suites (to account for randomness) generated with TAUC and with a random approach, which serves as a baseline. To be fair, we configured the random approach to generate timeliness scenarios with the same number of events in the test cases generated by TAUC. We considered test suites of various sizes, including 25, 50, 75, 100, and 122 test cases. The *BodySense* actual test suite contains 122 test cases and was therefore considered to represent a realistic test budget.

Modifying the implementation of *BodySense* to create faulty versions for purely experimental purposes and running all the test suites on all mutant implementations on the actual deployment platform is far too expensive. For this reason, we automatically generated 323 faulty versions of the Timeliness Test Model by means of dedicated mutation operators, then simulated the execution of the test cases against the mutated models by adapting the approach described in [Hessel et al., 2004] to our case.

Different mutation operators for timed automata are described in literature [Aboutrab et al., 2013, Aichernig et al., 2013]; for our experiments, we considered the mutation operators that may impact timing requirements, including Restricting Clock Conditions [Aboutrab et al., 2013], Widening Clock Conditions [Aboutrab et al., 2013], Shifting Clock Conditions [Aboutrab et al., 2013], Change Target Location (CTL) [Aichernig et al., 2013]. We did not consider operators that alter the functional behaviour only, for example, the ones that change the source and the target of an edge. However, to simulate the case in which the system is stuck in a state and breaks timing requirements, we configured the CTL operator to create self-loops on edges with clock variables.

We generated each faulty version of *BodySense* models by executing a single mutation operator on the Timeliness Test Model. All the guard conditions, invariants, and edges of the Timeliness

Table 4.3. Fault Coverage Measured Against 323 Faulty Versions.

	Average Coverage (± Std. Dev) by Test suite size (number of test cases)				
	25	50	75	100	122
TAUC	85%	88%	91%	91%	91%
	±0	±0.41	±0.46	±0.38	±0.42
Random	7%	12%	22%	30%	40%
	±0	±3.26	±3.16	±5.96	±3.58
Manual	-	-	-	-	60%

Test Model are mutated once by each applicable mutation operator. Mutation operators may lead to equivalent mutants, i.e. timed automata that satisfy the original requirements. We manually removed all the equivalent mutants and ended up with an evaluation benchmark of 323 mutant versions of BodySense timeliness test models.

Simulation of test cases execution is based on UPPAAL and requires that executable test cases are translated into sequential timed automata to be composed with the *BodySense* mutated models [Hessel et al., 2004]. The generated automata have edges that either send inputs to the system or validate operation results. A dedicated testing clock is used to trace execution time. Operation results are validated by means of guard conditions that check if system variables have specific values when the testing clock reaches a given value, i.e the time appearing in the wait conditions of the executable test cases. An error state is reached when the values are not the expected ones. A test case is considered to fail if an error state is reachable.

Results. We measured the percentage of faults (mutants) identified by each test suite. A test suite identifies a fault if at least one of its test cases fails. Table 4.3 reports the results obtained with the three approaches (for TAUC and random we report the average across the 10 runs). The table clearly shows that TAUC detects between two and twelve times more faults than random testing, depending on the test suite size. It is also clear that TAUC is significantly more effective than expertise-based manual testing with 31% additional faults detected, on average. In addition, TAUC is more effective than both random and manual testing even with much smaller-size test suites. This mostly results from the capability of TAUC to generate input sequences covering non-trivial interactions between system components. For example, to increase test diversity, TAUC generates test cases that send multiple messages on the bus thus enabling the detection of faults that slow down the execution of the interrupt handler, which in turn causes a delay in the qualification of errors.

TAUC subsumes the other approaches, i.e. TAUC detects all the faults detected by the random and the manual test suites. However, TAUC does not identify all the faults. Since, in the case of *BodySense*, TAUC has to rely upon partial oracles that check state variable values only and not the complete system state also captured by active locations. Such information, in embedded systems, is often not observable, as discussed in section 4.7. The faults not covered by TAUC are caused by the CTL operator, which leads to mutated edges that correctly update system variables but bring the system into a wrong state.

TAUC test cases always exercise the fault, i.e. cover the mutated edge, but only the test cases that verify additional system behaviors after reaching the faulty state can detect the fault (some of TAUC test cases do this by sending additional inputs to the system and by detecting that the system response is not the expected one). Test cases that terminate just after reaching the faulty state instead cannot fail. Manual and random test cases suffer from the same problem.

4.8.2 RQ2. Is the modelling and analysis effort required by TAUC acceptable in an industrial context?

In the case of TAUC, testing cost depends on the effort required for producing timed automata, use case specifications, and mapping tables.

To perform the experiment, there was no additional cost associated with writing use case specifications as they were already produced for communication purposes and to perform functional testing with UMTG.

We supported software engineers in the design of 25 timed automata: three environment automata (with a total of 6 locations, 11 edges, and 8 guard conditions) and 22 requirement automata that model error conditions. Note that modelling of requirement automata is simplified by the fact that the different errors are managed in similar ways (all the automata match the same template, which has 4 locations, 10 edges, 2 guard conditions and 2 scenarios events). Such numbers suggest that the complexity of the models required by TAUC can be managed by experienced software engineers. In addition, given that timed automata provide a clear representation of timing requirements, usually spread across textual specification documents, engineers agreed that their value goes beyond the benefits provided by automatic test generation.

4.8.3 Threats to Validity

Internal threats

Threats to the internal validity of the empirical observations we made may depend either on the components developed for generating the test cases. Our implementation includes third party components, more precisely the UPPAAL model checker (to derive a test suite that covers all the edges of the requirements automata), and a set of components that we developed to derive Timeliness Test Models, and generate test suites based on meta-heuristic search.

We assume that the implementation of UPPAAL is correct because it has been used in several research projects. To verify the correctness of the components developed by us, we manually inspected all the generated models to look for major errors, and, also, we inspected the generated test cases to identify major problems in the generated test suite (e.g., inputs not related to the scenarios covered by the test case).

External threats

Threats to the external validity regard the generalisability of our findings. We consider the test cases developed by IEE engineers being an example of carefully defined manual test cases. *BodySense* must comply with safety standards, and has already been installed on cars produced by several car manufacturers. Thus, we expect that the test cases provided by IEE engineers can test most of the foreseeable execution scenarios. For this reason we assume that our considerations about the effectiveness of TAUC test suites compared to manual testing might generalize to many systems.

4.9 Conclusion

In this Chapter we presented TAUC, a model-driven approach, inspired by industrial practice, to automate the system testing of software timeliness properties in the context of use-case driven development, when system requirements are expressed as use cases to communicate with clients and other stakeholders. In addition to use case specifications, TAUC relies on timed automata to capture timing requirements and relevant environment properties. It generates effective test cases by means of a meta-heuristic search algorithm that maximizes test suite diversity. Our objective is to minimize modelling overhead while enabling effective test generation in contexts where it cannot be guided by test execution results, e.g., embedded systems.

TAUC processes use case specifications to identify the test inputs that fire the state transitions in timed automata. It automatically identifies dependencies between use case specifications and timed automata, and captures them in timeliness test models that are also timed automata, thus enabling the use of UPPAAL for test automation. Timeliness test models are fed into UPPAAL to generate test cases that guarantee a basic coverage of edges with timing requirements.

To generate test cases that effectively stress timing requirements given a certain budget (test suite size), TAUC uses a meta-heuristic search algorithm that iteratively improves the test cases generated with UPPAAL to build a test suite that maximizes test case diversity, which has shown in past studies to increase fault detection.

Empirical results obtained with *BodySense* show that, given a constant test budget, TAUC is significantly more effective than a manual test suite devised by experienced engineers and random testing.

Chapter 5

Oracles for Testing Software Timeliness with Uncertainty

In this chapter, we address the problem of deriving automated oracles for testing software timeliness in the presence of non-deterministic behaviour caused by time uncertainty.

Although embedded systems are developed in a way to ensure their operations comply with pre-specified timing constraints, most of the timing specifications provided by software engineers are characterized by uncertainty [David et al., 2012]. A common example of uncertainty is given by timing constraints expressed as ranges (e.g., a software operation is expected to last between two and four seconds).

Time uncertainty in software specifications may be due to multiple reasons, for example, design choices. An example of design choice is the sampling of sensor values in a control loop at a given time rate. This case is very common in embedded and cyber-physical systems because many real-world events can be detected only by polling the environment in a control loop and, therefore, it cannot be determined with precision when the system will be able to observe and process them. This may, in turn, lead to non-deterministic system behaviors that make testing more difficult, and most particularly the definition of test oracles, as illustrated below.

A concrete example of uncertainty in timing specification is *BodySense*. This system implements a control loop that first samples temperature sensors, on a regular basis, and then performs other activities; this system is characterized by uncertainty since it may detect overheating with a maximum delay that corresponds to the duration of the loop execution. The specifications for this system will thus indicate that the system should be able to detect overheating within a certain time range. Such uncertainty in the timing specification may lead to non-deterministic software behaviors as the exact time at which the *BodySense* system reports overheating is not exactly known.

In this chapter we address the problem of building oracles for non-deterministic systems where the source of non-determinism is time uncertainty.

Most of the existing work on testing the timing properties of software systems focuses on schedulability analysis for real-time systems [Davis and Burns, 2011]. These approaches address a different problem than ours, i.e., the development of techniques to test or verify that software tasks are able to

meet their deadlines.

The few approaches that address the problem of testing software timeliness in the presence of uncertainty are based on online testing [Krichen and Tripakis, 2004, Larsen et al., 2005, David et al., 2012]. Online testing approaches rely upon models of the expected software behaviors (e.g., timed automata) to determine the test inputs to send to the system at runtime, during test case execution, based on the observed outputs. Unfortunately, online testing approaches might have limited applicability because they require that the testing framework provides programmable APIs, which are not always available, to build a test adaptation layer. In addition to this, the test adaptation layer itself may, in turn, introduce latency and thus be inappropriate for testing real-time systems.

An additional limitation is that most existing timeliness testing approaches do not take into account stochasticity; more specifically, they do not verify if the frequency of appearance of the expected outputs comply with the specifications, nor do they verify if the observed results can give statistical guarantees for the validity of the test outcome. The effects of stochasticity are taken into account by solutions [Ševčíková et al., 2006, Yoo, 2010, Patrick et al., 2016, Hierons and Merayo, 2009], which rely upon statistical tests to determine if the output of a stochastic system follows an expected distribution. Among these, the only approach targeting timing requirements [Hierons and Merayo, 2009], cannot be applied in our context because it focuses on the generation of complete test suites, and cannot be adopted to generate oracles for test suites derived with different criteria.

In this work, we focus on systems where online testing is not an option, and we assume that the timing constraints of the system are specified by means of timed automata. The type of uncertainty that we consider in this chapter is common in industrial settings and relates to time triggered transitions (i.e., internal transitions that are enabled when a constraint on clock variables evaluates to true and do not need to synchronize with any event), which are a typical cause of uncertainty [David et al., 2012].

Traditional approaches that derive test cases from finite state machine (FSM) specifications rely upon Unique Input-Output (UIO) sequences to derive effective test oracles [Lee and Yannakakis, 1996, Derderian et al., 2006]. The adoption of UIO sequences as oracles is motivated by the lack of observability that usually characterizes certain types of systems, including embedded systems. Unfortunately, the existing approaches for deriving UIO sequences from FSM and EFSM, e.g. [Robinson-Mallett et al., 2006], do not work with timed automata and, more specifically, cannot deal with time-triggered transitions. In addition, the non-deterministic behaviour caused by time uncertainty affects the reliability of oracles derived from UIO sequences. In the presence of time uncertainty, multiple valid states could be legally reached after a given input sequence, which means that multiple, perfectly legal output sequences might be observed for a same sequence of test inputs, making the detection of a fault more challenging.

To address the above challenges, in this chapter, we propose Stochastic Testing with Unique Input-Output Sequences (*STUIOS*), an approach for the automated generation of effective oracles for timeliness test cases derived from timed automata, in the presence of uncertainty.

In this chapter, we introduce the concepts of *stochastic test cases* and *probabilistic UIO sequences* (PUIO sequences). A PUIO sequence is an input-output sequence with an associated probability of observing the given output sequence in response to the inputs. The underlying idea is that probabilistic

UIO sequences enable fault detection by determining if the output sequences observed through testing are unlikely, based on multiple executions of the same test cases. Stochastic test cases extend the same idea to the entire test case. A stochastic test case specifies the expected probability of observing a specific output sequence after a given test input sequence; in addition, it includes a PUIO sequence that is used to check if the test execution has likely brought the system to the expected state.

STUIOS receives as input the timing specifications of the system expressed as a network of timed automata, and a set of test cases derived from these specifications either manually by the software engineers, or automatically. *STUIOS* generates stochastic test cases that include the input-output sequences from the test cases provided by software engineers, and automatically derives corresponding expected probabilities and PUIO sequences.

One significant contribution and innovation is that *STUIOS* combines statistical model checking and UIO techniques adapted to timed automata to address the challenges stated above. *STUIOS* generates stochastic test cases by relying on the simulation capabilities of the UPPAAL model checker and by automatically identifying UIO sequences from timed automata. Further, it derives probabilities for both the identified UIO sequences and the provided test output sequences based on statistical model checking software, in this dissertation the statistical extension of UPPAAL. Finally *STUIOS* identifies faults by repeatedly executing test cases with the associated PUIO sequences and making use of statistical hypothesis testing to determine the test verdicts (pass or fail) and decide when to stop test case executions.

The contributions of this chapter can be thus summarized as follows:

- We define the concepts of *stochastic test cases* and *probabilistic UIO sequences*, and explain how can they be adopted to address the problems stated above.
- We propose *STUIOS*, a technique that generates stochastic test cases after processing the timing specifications of the system expressed as a network of timed automata and a set of test cases derived from these specifications.
- We report an empirical evaluation of the approach using *BodySense* as a case study.

This chapter is structured as follows. Section 5.1 is a background section that introduces the notation used in this chapter to model test cases. In section 5.2 we motivate the work presented in this chapter by discussing the issues to expect when generating oracles from timed automata with uncertainty. Section 5.3 presents the *STUIOS* approach and its steps. Section 5.4 reports on the empirical results obtained from an industrial case study in the automotive domain. Section 5.5 concludes the chapter.

5.1 Modelling of Test Cases

This section introduces the notation used in this chapter to model test cases.

In the presence of software specifications elicited by using FSMs, software engineers derive test cases according to criteria whose objective is to maximize the probability of finding faults. A commonly adopted criterion for example is edge coverage, which consists of generating test cases so that all the edges of the timed automata are covered, at least once, during test execution.

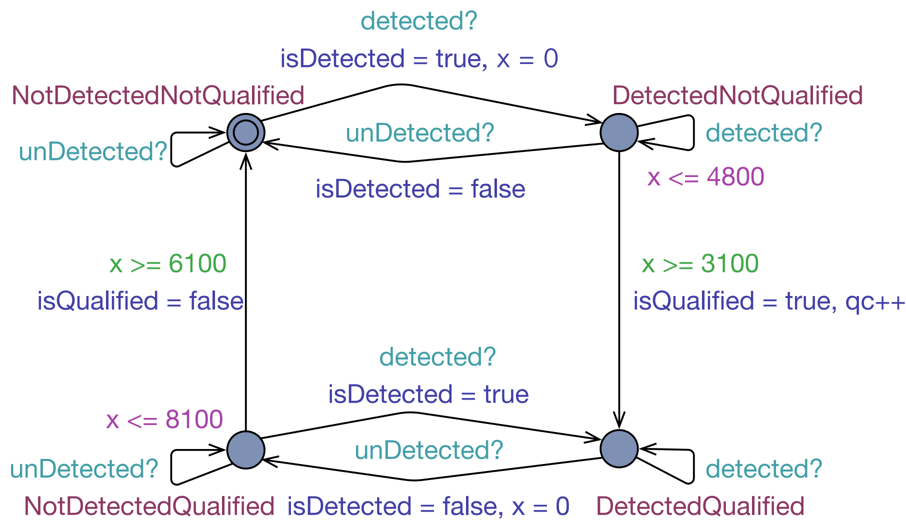


Figure 5.1. Automaton that captures how temperature errors are qualified and dequalified in *BodySense*.

We model test cases derived from TAs as sequences of input / output pairs. Each pair indicates the output expected to be generated by the system after a given input. In embedded systems, inputs and outputs typically correspond to system events. However, since TAs include state variables, the expected output indicated in a test case also includes the expected value of observable state variables. In addition, since edges in TAs might be triggered by time delays, inputs might be expressed as wait operations.

An example test case is shown in Figure 5.2. This test case is derived from the TA in Figure 5.1 and checks if temperature errors are qualified on time. This test case covers the edges of the shorter path from location `NotDetectedNotQualified` to location `DetectedQualified`, and checks if state variables and outputs contain the expected result after each input or time delay. The first line of the test case indicates that after the event `detected` is observed by the system, then the system should set its observable state variables `isQualified`, `isDetected` and `qc` to `false`, `true`, and `zero`, respectively.

```
<detected>/ (isQualified == false, isDetected == true, qc == 0)
<wait for 4800 ms> /  (isQualified == true , isDetected == true, qc == 1)
```

Note: We use the symbol / to separate inputs from the expected results (i.e., the values of the observable state variables that should be verified as test oracle).

Figure 5.2. A Test Case that checks if TemperatureErrors are qualified on time.

5.2 Testing with UIO Sequences

The state of the actual software system that implements the timing specifications expressed in TAs is partially observable. More specifically, we expect that the software cannot be queried to determine which are the active locations in the abstract representation of its state, furthermore, not all the state variables used in the TAs have an observable counterpart in the concrete implementation of the system.

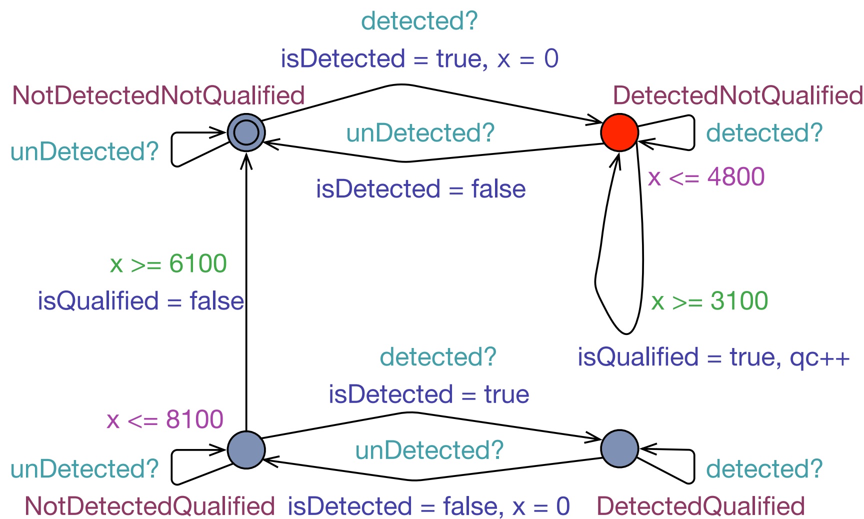


Figure 5.3. Automaton that shows a possible transition fault in the implementation of Figure5.1.

The limited observability which characterizes TAs usually makes oracles that simply check if the expected output sequence has been generated after test inputs ineffective. These oracles for example do not detect transition faults which are faults that manifest when the active location of a TA following a state transition is incorrect but the system updates its state variables and generate outputs as expected (this problem has been observed during the experiments reported in Chapter 4 Section 4.8.1).

In the case of *BodySense* a transition fault may keep the system in location `DetectedNot-Qualified` for more than the allowed time (4800 ms). Location `DetectedQualified` is expected to become active if a temperature error remains detected in the system for 4800 ms. Figure 5.3 provides a model of the behaviour triggered by this fault. This fault cannot be detected with the test case in Figure 5.2 which simply checks the values of the observable state variables `isDetected` and `isQualified` after waiting for 4800 ms, because these two variables are updated as expected.

When specifications are given in the form of FSMs, common approaches for testing in the presence of partial observability are based on the identification of Unique Input-Output (UIO) sequences [Lee and Yannakakis, 1996, Derderian et al., 2006]. A UIO sequence is composed of a sequence of inputs and corresponding outputs generated by the system. It provides the guarantee that the expected output sequence is observed only when the system is in a specific known state. An oracle derived from a UIO sequence is executed after a test case, and checks if the system is in the expected state by sending the inputs specified by the UIO sequence and by verifying that the system responds with the expected output sequence. We model UIO sequences in the same way as test cases, i.e. by using sequences of IO pairs that include time delays and observable state variables.

5.2.1 Generation of UIO sequences from TAs

We model UIO sequences in the same way as test cases, i.e. by using sequences of IO pairs that include time delays and observable state variables.

One of the aspects to consider when generating UIO sequences from TAs is context dependency.

This characteristic is typical of UIO sequences derived from FSMs including state variables (e.g., EFSM), such as clocks in the case of TAs. Context dependency implies that the path specified by a UIO sequence might be feasible only for specific values taken by state variables, which in turn implies that the output of a UIO sequence depends on the values of state variables [Ramalingom et al., 1996]. Context dependency cannot be ignored; a simple yet effective solution to deal with context dependency that we include in our approach is to generate a different UIO sequence for each test case of the test suite.

```
<undetected> /  ( isQualified == true, isDetected == false, qc == 1)
<wait for 8100ms> /  ( isQualified == false, isDetected == false, qc == 1 )
```

Figure 5.4. UIO sequence that characterises state DetectedQualified after the execution of the test case in Figure 5.2.

Figure 5.5 shows the test case of Figure 5.2 followed by a UIO sequence (blue lines) to determine if the system has reached location `DetectedQualified` after the execution of the test.

```
<detected>/ ( isQualified == false, isDetected == true, qc == 0)
<wait for 4800 ms> /  ( isQualified == true , isDetected == true, qc == 1)
<undetected> /  ( isQualified == true, isDetected == false, qc == 1)
<wait for 8100ms> /  ( isQualified == false, isDetected == false, qc == 1 )
```

Figure 5.5. Test case of Figure 5.2 including the UIO sequence in Figure 5.4.

The first line of the UIO sequence in Figure 5.5 indicates to send the input event `undetected` to the system (i.e., bring the temperature back to normal). As a consequence, the system should make location `NotDetectedQualified` active (this cannot be observed by a tester), and set the variables `isDetected`, `isQualified`, and `qc` to `false`, `true` and `one`, respectively. This IO pair can distinguish location `DetectedQualified` from locations `NotDetectQualified` and `NotDetectNotQualified` since in both cases the value of `isDetected` would not have changed to `false`.

The second line indicates to wait for 8100 ms and then check that the variables `isDetected`, `isQualified`, and `qc` are set to `false`, `false` and one respectively. This last IO pair allows to distinguish location `DetectedQualified` from location `DetectedNotQualified`. In the case location `DetectedNotQualified` is active in place of `DetectedQualified` after test case execution, then the event sent by the first UIO line has brought the system in location `NotDetectedNotQualified` without changing the value of variable `isQualified` (the first line of the UIO thus do not allow to distinguish the two locations). The second line of the UIO then, makes the system wait for 8100 ms, and after this delay, the system remains in the same location (`NotDetectedNotQualified`) without changing its state variables, thus distinguishing the two locations.

We can notice that the UIO sequence is context dependent because although it can be used to check if the execution of the test case in Figure 5.2 has brought the system to location `DetectedQualified`, it cannot be used to check if other test cases terminate in the same state. For instance, this UIO sequence cannot be used to check if location `DetectedQualified` has been reached after executing a test case that traverses the sequence of locations `DetectedNotQualified`,

`DetectedQualified`, `NotDetectedNotQualified`, `NotDetectedQualified`, and `DetectedQualified`. In this case, the expected value of the state variable `qc` will be `two`, not `one` as indicated in the UIO sequence of Figure 5.4.

In addition to accounting for context dependency, to effectively support testing, we need to automatically generate UIO sequences. Unfortunately, to the best of our knowledge, there are no approaches focused on deriving UIO sequences from TAs characterised by uncertainty. In the case of TAs without uncertainty, existing approaches for generating UIO sequences from EFSM could be applied, because of the similarity between EFSM and TA formalisms, but these approaches cannot be adopted in the presence of uncertainty. For example, the approach proposed by Robinson et al. [Robinson-Mallett et al., 2006] requires deterministic EFSM with transitions that are always triggered by some input event and thus cannot be applied to TAs characterised by uncertainty because they present time-triggered transitions.

5.2.2 Consequences of uncertainty

In the following paragraphs, we report three critical aspects that should be considered when generating test oracles for test cases derived from TA: (1) uncertainty may limit the fault detection effectiveness of test oracles; (2) uncertainty may lead to multiple valid test outcomes whose frequency of appearance must be checked to verify if the software complies with the specifications; (3) uncertainty may lead to scalability problems during testing because of an unmanageable number of oracles in a test suite.

Uncertainty limits the fault detection effectiveness of test oracles

One of the effects of uncertainty is non-determinism, and this may limit the effectiveness of test oracles in the presence of faults that affect the implementation of timing constraints.

Consider for example a faulty implementation of *BodySense* that may take up to 6000 ms to qualify an error. The test case in Figure 5.5 is able to detect the fault only when *BodySense* takes more than 4800 ms to qualify an error during test execution. If the fault manifests itself in a non-deterministic way, a single test case execution may not be able to detect its presence. When testing timing requirements, multiple test case executions might be needed to obtain statistical guarantees about the correctness of results.

Oracles must take into account the frequency of expected output

A consequence of uncertainty is that same input may bring the system into multiple valid locations, with the consequence that a single test case execution may lead to multiple valid results. This implies that the failure of an oracle that simply verifies if the system has reached a specific state (or, has generated a specific output sequence) may not indicate the presence of a fault.

An example of the effect of uncertainty on test oracle reliability is given by a test case that checks if the system behaves properly when a temperature error appears just for a short time. A test case designed for this purpose is shown in Figure 5.6. This test case checks if the system is able to detect a temperature error that remains active just for 4000 ms, qualify it, and then de-qualify it. If *BodySense*

```
<detected> / (isQualified == false, isDetected == true, qc == 0)
<wait for 4000ms> / (isQualified == true, isDetected == true, qc == 1)
<undetected> / (isQualified == true, isDetected == false, qc == 1)
<wait for 8100 ms> / (isQualified == false, isDetected == false, qc == 1)
<detected> / (isQualified == false, isDetected == true, qc == 1)
```

Figure 5.6. Test case that verifies the qualification of a temperature error that appears for a short time.

does not qualify the temperature error between 3100ms and 4000ms, the test case in Figure 5.6 will fail when checking the condition `isQualified==true, isDetected==true, qc==1`.

A single failure of this test case does not indicate the presence of a fault in *BodySense* since the specifications indicate that *BodySense* can take more than 4000 ms to qualify an error. However, if the test case never generates the expected output, then this can be an indication of a problem since the specifications indicate that the system may qualify errors between 3100 ms and 4000 ms.

More complex examples can be observed in the presence of dependencies between different TAs. In the case of *BodySense*, this is observed with test cases covering errors with different qualification priorities. For example, a sensor error cannot be qualified if a temperature error has already been qualified. In these cases, testing is complicated by the effect of timing uncertainty on the interplay between the state of different automata. For example, a test case that checks if both sensor and temperature errors are qualified within the timing bounds (i.e., 4800 ms) may thus fail depending on which error is qualified first, but in general, engineers expect that such a test case passes according to a given frequency that depends on the timing ranges appearing in the model.

To correctly evaluate test results, we thus need to know the expected probability that the system responds with a specific output sequence. In the presence of uncertainty, an implementation is faulty if and only if, over multiple executions, the various possible outputs are not observed with their expected frequency. This observation shows the need for oracles to take into consideration the probability of observing a given output in order to determine if a test case passes or fails. We will refer to such test cases as *stochastic test cases* in this work.

Uncertainty may lead to scalability problems

Because of non-determinism, the execution of a test case might correctly bring the system into multiple states, and, consequently, generate multiple valid output sequences. However, verifying that the system can reach all the valid states after a test execution (i.e., generate all the valid outputs), might easily lead to scalability issues. This may be the case if each test case needs to be executed multiple times, until every expected output is observed, including the least probable ones.

To deal with this problem, and thus find a balance between testing cost and test cases effectiveness, we assume that engineers are interested in verifying if the system has been able to reach a single specific location that we refer as *desired final test location*, which is the location that should be always reached if the system was deterministic (e.g., if the time triggered transitions are fired at a specific time). Usually this is the location that should be reached with the highest probability. In the case of the test case in Figure 5.6, for example, software engineers are interested in verifying if location *DetectedQualified* is active after detecting a temperature error and waiting for 4000 milliseconds. Location *DetectedQualified* is the location that should be always reached if the system was deterministic, for example if the edge between locations *DetectedNotQualified* and *DetectedQualified* was triggered when the clock reaches 4000.

5.3 The *STUIOS* approach

5.3.1 An Overview

Figure 5.7 shows an overview of the *STUIOS* approach. Note that all the values in that figure are hypothetical and are there just to convey the idea underlying the approach. *STUIOS* works in two phases: (1) it *generates stochastic test cases* by extending the test cases provided by software engineers; (2) it *executes the stochastic test cases and attempts to determine whether they pass or fail.*

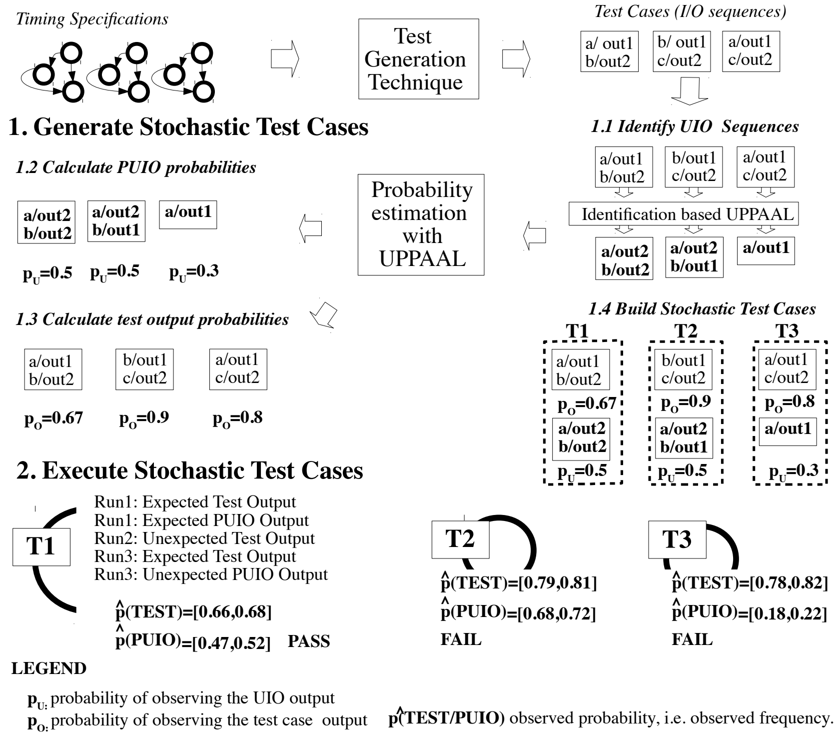


Figure 5.7. An overview of the *STUIOS* approach

STUIOS receives as input a test suite (in the form of IO sequences), which we assume to be derived from specifications elicited as a network of TAs. Any test criterion can be used to generate the test suite. We simply expect that each test case of the test suite is defined as an IO sequence (i.e., a sequence of pairs <*input action*> / <*expected output*> that matches a feasible path of the automaton, as described in Section 5.1). An example test case is shown in Figure 5.6.

STUIOS wraps the given test cases into stochastic test cases that handle non-determinism with the aim of reducing incorrect failures.

For each test case, we also expect that software engineers specify a single location (with highest probability where the system states) that is expected to be checked for being active after the test case execution (this is the *desired final test location*).

We define a stochastic test case as a test case that includes an IO sequence that covers a specific test scenario, followed by a PUIO sequence that checks if the software has reached the desired final

test location. A stochastic test case specifies both the probability of observing the test scenario output and the probability of observing the output of the PUIO sequence.

To *generate stochastic test cases*, *STUIOS* performs four steps (shown in Figure 5.7). In *Step 1.1*, *STUIOS* identifies a UIO sequence for each final test location and test case provided by software engineers. In *Step 1.2*, *STUIOS* builds corresponding PUIO sequences, which means that it calculates, for each UIO sequence, the probability of observing its output when the UIO sequence is executed after the corresponding test case. In *Step 1.3*, *STUIOS* calculates the probability of observing the output expected by each test case. Steps 1.1 to 1.3 rely on the simulation and verification capabilities of the UPPAAL model checker [Bengtsson et al., 1995].

In *Step 1.4*, *STUIOS* builds stochastic test cases. Each generated test case specifies the probability of observing the output sequence of the provided test case, and includes the PUIO sequence that checks if the desired final test location has been reached.

In the second phase (*Execute Stochastic Test Cases* in Figure 5.7), *STUIOS* executes each stochastic test case against the software implementation multiple times (until we have statistical guarantees about the test outcome, as discussed below) and collects test results. *STUIOS* relies upon statistical hypothesis testing to decide when an accurate test result can be drawn from multiple executions of the same test case. Test execution is stopped when we have sufficient statistical guarantees that the test cases either pass of fail. More specifically, a test case fails if the test case output or the PUIO output are not as frequent as expected, considering a confidence interval, and otherwise passes.

After a brief overview of the capabilities of UPPAAL exploited by *STUIOS*, the following sections provide additional details about the activities performed by *STUIOS*.

5.3.2 Probability Estimation with UPPAAL

To generate PUIO sequences (Steps 1.1 and 1.2 in Figure 5.7) and calculate test case probabilities (Steps 1.3), *STUIOS* relies upon UPPAAL [Bengtsson et al., 1995]. UPPAAL is a model checker for network of TAs that provides model checking and statistical model checking capabilities. *STUIOS* more specifically makes use of the verification of properties (e.g., reachability) expressed using computation tree logic (CTL), the simulation of the model execution, and the estimation of the true probability of a model property (i.e., the probability that a property holds).

We briefly describe the probability estimation feature of UPPAAL as some readers may not be familiar with it. UPPAAL can estimate the probability that a given property of the system holds within a certain time frame, e.g., it can estimate the probability that a single location of a network of TAs is reached in 4000 clock ticks. UPPAAL relies upon an algorithm that resembles Monte-Carlo simulation to compute the probability estimation. Additional details are described in [Hérault et al., 2004].

5.3.3 Generating Stochastic Test Cases

We now describe the steps followed by *STUIOS* to generate stochastic test cases (Step 1 in Figure 5.7). Figure 5.8 shows the activities for generating stochastic test cases, in an algorithmic form.

Require: TS, the test suite
Require: TAS, the timing specifications (network of TA)
Require: ftL, the final test location
Require: L, the maximum length of a UIO sequence
Ensure: STS, a stochastic test suite derived from TS

1: **for** $testCase\ in\ TS$ **do**
2: //1.1 Identify a PUIO for testCase
3: //1.1.1 Determine the context of the PUIO
4: $testTA \leftarrow generateTestAutomaton(testCase)$
5: $trace \leftarrow reachabilityAnalysis(TAS, testTA)$
6: $stateVariableAssignments, activeLocations \leftarrow$
7: $extractContextInformation(trace)$
8: //1.1.2 Identify potential UIO sequences
9: $pathLen \leftarrow 1$
10: $uioFound \leftarrow FALSE$
11: **while** $pathLen < L$ && $uioFound == FALSE$ **do**
12: $potentialUIOS \leftarrow$
13: $depthFirstVisit(TAS, stateVariableAssignments, activeLocations)$
14: **for** $PUIO\ in\ potentialUIOS$ **do**
15: $uioTA \leftarrow generateTestAutomaton(PUIO)$
16: $locations \leftarrow activeLocations \setminus ftL$
17: $TA_{ftL} \leftarrow retrieveAutomatonContaining(ftL)$
18: $otherLocations \leftarrow allLocationsIn(TA_{ftL}) \setminus ftL$
19: $isUIO \leftarrow TRUE$
20: **for** $oL\ in\ otherLocations$ **do**
21: $newActive \leftarrow locations \cup oL$
22: $trace \leftarrow reachabilityAnalysis$
23: $(TAS, uioTA, newActive, stateVariableAssignments)$
24: **if** $trace \neq null$ **then** **//final state is reachable**
25: $isUIO \leftarrow FALSE$
26: *break*
27: **end if**
28: **end for**
29: **if** $isUIO == TRUE$ **then**
30: **//1.2 Build PUIO sequence**
31: $timeBound = sumAllTheDelaysIn(uioTA)$
32: $p_{PUIO} \leftarrow probabilityAnalysis(TAS, uioTA,$
33: $timeBound, activeLocations, stateVariableAssignments)$
34: **//1.3 Calculate test output probability**
35: $timeBound = sumAllTheDelaysIn(testTA)$
36: $p_{test} \leftarrow$
37: $probabilityAnalysis(TAS, testTA, timeBound)$
38: **//1.4 Build stochastic test case**
39: $stochasticTest \leftarrow < testTA, p_{test}, PUIO, p_{PUIO} >$
40: $STS \leftarrow STS \cup stochasticTest$
41: **end if**
42: **if** $isUIO == TRUE$ **then //A valid UIO has been found**
43: break **//Terminate the iteration for the current test**
44: **end if**
45: **end for**
46: $pathLen \leftarrow pathLen + 1$
47: **end while**
48: **end for**

Function *reachabilityAnalysis* performs reachability analysis with UPPAAL by processing a network formed by the software specifications and the test automaton.
Function *probabilityAnalysis* returns the probability of reaching the final state of the given test automaton in the provided time bound as calculated by UPPAAL.
The additional parameters appearing in lines 23 and 33 are used to set the initial state of the system (otherwise the initial state is specified in the software specifications).

Figure 5.8. The algorithm to generate stochastic test cases.

Identify UIO Sequences (Step 1.1 in Figure 5.7)

To generate UIO sequences *STUIOS* performs three activities: (1) it determines the context (active locations and value of state variables) of each UIO sequence, (2) it identifies all the potential UIO sequences for a given final test execution state, (3) it identifies a valid context-dependent UIO sequence among the potential UIO sequences found (i.e., it ensures that the UIO sequence generates a unique output to detect transition faults).

Activity 1 of 3-Determining the context of a UIO sequence UIO sequences for TAs are context dependent (as discussed in Section 5.2.1). Since *STUIOS* uses UIO sequences to determine if the execution of a test case has brought the system into a given location (i.e., it executes a UIO sequence after a corresponding the test case), the context of a UIO sequence coincides with the state of the system after the execution of test cases (hereafter final test state). In TAs the state of the system is given by both the active locations and the values of the state variables.

Though the final test state can be deducted by simulating test case execution against the TA network, *STUIOS* relies upon reachability analysis to speed up the process.

STUIOS first translates the IO sequence into a sequential timed automaton (hereafter test automaton, see Line 4 in Figure 5.8). The test automaton resembles the test case: each edge either sends a test input or checks if the output corresponds to test expectations. In case the output does not correspond to the expectations, an error state is reached, otherwise the computation continues. Figure 5.9 shows a test automaton derived from the test case in Figure 5.2. The edge between locations *L1* and *L2* captures the sending of the event *detected*, the edge between locations *L3* and *L4* simulates a delay of 4800 ms, while the other edges check for the expected value of observable variables, and activate location `Error` in case the values are not the expected ones.

STUIOS then exploits the reachability analysis feature of UPPAAL to identify a trace that shows the effects of the test case execution (Line 5 in Figure 5.8). This is done thanks to a reachability formula that specifies that the desired final test location should be reached after the test case execution (i.e., `E<> test.FinalTestLocation && te.DetectedQualified` in our running example). The trace generated by UPPAAL shows the value of all the state variables of the system after the execution of the test case and the list of active locations.

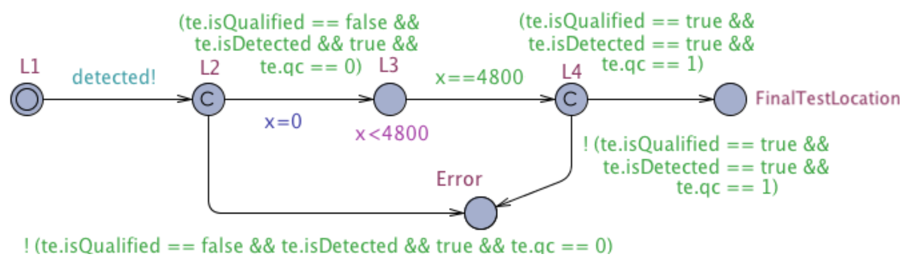


Figure 5.9. Test Automaton derived from the test case in Figure 5.7.

Activity 2 of 3-Identifying all the potential UIO sequences To identify all the potential UIO sequences of a specific length, *STUIOS* explores all the feasible paths of the timed automata by taking advantage of the UPPAAL simulator.

STUIOS explores all the feasible paths up to a length L given by the software engineer. The exploration is performed in a depth-first manner starting from the final test state (Line 7 in Figure 5.8). The paths traversed during the visit are *potential UIO sequences* since they capture the output sequence expected following additional inputs being sent to the system after test case execution, but do not give any guarantee about uniqueness.

The process for finding potential UIO sequences starts with a path length equal to 1 (Line 9 in Figure 5.8) and then the process iterates by increasing the value of the path length by one (Line 46) if none of the recorded sequence is recognized a valid UIO sequence, as explained in the following sections.

Activity 3 of 3-Identification of valid UIO sequences A valid UIO sequence guarantees that the expected output sequence is observed only if the test case has terminated in the desired final test location. Thus, a potential UIO sequence cannot be a valid UIO sequence if it generates the same expected output when the final test location is not the expected one. For this reason, *STUIOS* identifies valid UIO sequences by looking for a potential UIO sequence whose inputs never lead to the expected output sequence when the active location is different from the desired final test location.

STUIOS identifies a valid UIO sequence by iteratively processing all the potential UIO sequences. First *STUIOS* translates each *potential UIO sequences* into a sequential automaton (hereafter UIO automaton) by following the same process adopted to derive test automata (Line 15 in Figure 5.8).

Then *STUIOS* relies upon the reachability analysis feature of UPPAAL to determine if the UIO sequence can generate the expected output even when the test case terminates in a location different than the desired final test location (Line 24). Because we want to determine if the potential UIO sequence generates a different output when the final test location is not the expected one, *STUIOS* starts reachability analysis from the same active locations and variable values as those of the test termination state, except for the desired final test location. *STUIOS* iterates the process multiple times so that all the locations of the timed automaton containing the final test location are considered (see lines 16 to 20).

STUIOS identifies a UIO sequence when the final location of the test automaton is never reached. *STUIOS* repeats the process for all the potential UIO sequences till a valid UIO sequence is found (see the break instruction in Line 43).

5.3.4 Building PUIO Sequences

In the next step (Step 1.2 in Figure 5.7), *STUIOS* associates to each UIO sequence a value that captures the probability that the system responds with the expected output after the input sequence (Lines 31 to 33 in Figure 5.8). As discussed earlier, to calculate this probability, *STUIOS* uses the probability estimation feature of UPPAAL.

Since the final location of the test automata is reached only if the output expected by the UIO sequence is observed, *STUIOS* computes the probability of the UIO sequence by estimating the probability of reaching the final location in the test automata in the given test execution time. The test execution time is computed by summing all the delays in the test automata (see variable `timeBound` in Line 31).

Probability estimation is performed against a test automata that joins the test case automata with the UIO sequence test automata (Line 33).

5.3.5 Identifying Test Case Probabilities and Building Stochastic Test Cases

STUIOS computes the probability of observing the output expected by the test case (Step 1.3 in Figure 5.7) by following the same approach described in Section 5.3.4 (i.e., by computing the probability of reaching the final location of the test automaton, which does not include the PUIO sequence, see Lines 35 to 37 in Figure 5.8).

A stochastic test case is then built (Step 1.4 in Figure 5.7) using the PUIO sequences and the probability of observing the expected test output (Line 39).

5.3.6 Test Execution

STUIOS executes each stochastic test case in a loop till a stopping condition is met (Step 2 in Figure 5.7). We have identified three different stopping conditions: (1) the actual output is illegal (in this case, the test case fails), (2) the expected output occurs with the expected frequency (in this case, the test case passes), (3) the expected output does not occur with the expected frequency (in this case too, the test case fails). We describe in the following the three cases in more detail.

Case 1: The output is illegal. The output of a test case is illegal if the observed output sequence cannot be generated by the network of TAs that capture the software specifications, in other terms there is no path in the TAs that lead to the observed output sequence after the given input sequence.

The output sequence in Figure 5.10 shows an example of an illegal output for the TA in Figure 5.1. This outputs sequence is illegal if it is generated in response to the test inputs of the test case in Figure 5.6. The specifications for *BodySense* indicate that the system, after waiting for 4000 ms (second line of the test case), is allowed to either stay in location `DetectedNotQualified`, thus keeping the variables `isQualified` and `qc` set to `false` and `0` respectively, or move to location `DetectedQualified`, and set the values of the two variables to `true` and `1` respectively (these are the values expected by the test case Figure 5.6). The faulty output in Figure 5.10 shows that the system sets the value of variable `isQualified` to `true` but keeps variable `qc` set to `zero` after waiting for 4000 ms. This combination of values is not a legal output for the TA in Figure 5.1.

STUIOS thus stops executing a test case if it leads to an invalid output sequence because this clearly indicates that the software does not meet its specifications. The test case is considered to have failed.

```
(isQualified == false, isDetected == true, qc == 0)
(isQualified == true, isDetected == true, qc == 0)
```

Figure 5.10. Portion of a faulty output sequence generated after the execution of the test case in Figure 5.6.

Cases 2 and 3: The output is valid. If the system produces legal outputs (i.e., an output that can be generated according to the software specifications), then we need to determine if, across multiple executions of the stochastic test case, the system generates, among all the legal output sequences generated, the output sequence expected by the stochastic test case with the expected frequency. Since

stochastic test cases include both test cases provided by software engineers and PUIO sequences, we need to observe that both the following output sequences occur with the expected frequency: (1) the sequence of output values (hereafter $output_{TEST}$) generated after the inputs belonging to the original test case and (2) the output sequence (hereafter $output_{PUIO}$) generated after the inputs of the PUIO sequence.

We assume that each execution of a test case is independent; this assumption generally holds for stateless systems and for systems that provide means to reset the system state between the executions of different test cases (this the case of *BodySense*). Since each execution of a test case is independent and since the result is either pass or fail, we apply hypothesis testing based on binomial distribution (binomial test) to determine if the system behaves according to specification. A test case fails if the null hypothesis “the expected output occurs with the expected frequency" is rejected.

Given the observed probability, the expected probability, and the significance level α (which captures the desired likelihood of encountering a Type I error[1], in our case 0.05), a hypothesis test indicates if the null hypothesis should be rejected. In our context, this means that the hypothesis test indicates if the system under test should be considered faulty.

To apply hypothesis testing, we rely upon the Wilson score interval because it is known to be reliable even in the presence of a small number of samples (i.e., executions of a test case in our context) [Zou et al., 2009].

The Wilson score interval is computed according to the following formula:

$$\frac{1}{n+z^2}\left[nS+\frac{1}{2}z^2 \pm z\sqrt{\frac{1}{n}*nS*nF+\frac{1}{4}z^2}\right]$$

where z is known as standard score, and depends on the value of α (for α=0.05 z=1.96), n is the total number of runs considered, nS is the number of successes (in our case the output of the test case is the expected one), nF is the number of failures (in our case the output of the test case is not the expected one). The confidence interval indicates that the estimated parameter has a probability of 1-α of lying in it (0.95 with α=0.05). A null hypothesis is thus rejected if the expected probability (in our context this coincides with the expected frequency of an output) lies outside this interval.

Hypothesis testing helps us determine if the system is faulty. However, to provide trustable results we would like to collect enough samples, i.e. repeat the execution of a test case for a sufficient number of times, to be confident about the outcome. Generally speaking, the more observations, the higher the confidence about the absence of faults. We determine if we have collected enough samples (i.e., executed enough runs of a same test case) based on the confidence interval. With a small interval, when the null hypothesis is not rejected, we can conclude that the estimated parameter (i.e., the frequency of occurrence of the expected test output) is close to the expected value generated with UPPAAL, with a given degree of confidence (0.95 in our example). Based on this observation we stop testing and assume that a test case passes when the null hypothesis is not rejected and the confidence interval is small enough. In this case we chose to have an interval of size 0.1 so that we have a 95% chance for the actual probability to be within a maximum distance of 0.1 from the probability determined from the specifications. For the same reason, we indicate that a test case fails

[1]Type-I errors consist of erroneously rejecting the null hypothesis

if the null hypothesis is rejected and the confidence interval is 0.1.

Since *STUIOS* applies hypothesis testing twice for each test case execution, considering both $output_{TEST}$ and $output_{PUIO}$, *STUIOS* proceeds as follows. When the confidence interval is smaller than 0.1 and the null hypothesis is supported for both $output_{TEST}$ and $output_{PUIO}$, then *STUIOS* indicates that the test case has passed. When the confidence interval is smaller than 0.1 and the null hypothesis is rejected in the case of either $output_{TEST}$ or $output_{PUIO}$, then *STUIOS* indicates that the test case has failed. Otherwise (i.e., if the confidence interval never becomes smaller than 0.1), the execution of the test case continues till a maximum number of executions is reached (we use 100 in our experiments). In this case, we indicate that the test case passes if the null hypothesis is confirmed for both $output_{TEST}$ and $output_{PUIO}$ and otherwise the test case fails, though in this case we also indicate that the test case outcome might not be reliable.

For example, the test case T1 in Figure 5.7 passes because the Wilson score interval calculated for $output_{TEST}$ after multiple runs of T1 is $[0.66, 0.68]$, which is smaller than 0.1 and includes the frequency of the expected output (i.e., 0.67), and the Wilson score interval calculated for $output_{PUIO}$ includes the expected frequency as well (i.e., 0.5). The test case T2 instead fails because the Wilson score interval calculated for both $output_{TEST}$ and $output_{PUIO}$ does not include the expected values. Finally the test case T3 fails because the Wilson score interval calculated for $output_{PUIO}$ does not include the expected value.

5.4 Empirical Evaluation

To evaluate the effectiveness of the *STUIOS* approach, we conducted an empirical evaluation, based on an industrial case study, addressing the following three research questions:

- *RQ1. How does* STUIOS *compare with a simpler and less expensive alternative?* This research question aims to determine whether *STUIOS* performs significantly better than a much simpler and less expensive alternative that derives test cases that do not include UIO sequences and whose oracles rely solely on analyzing output sequences (referred to below as "simple oracles").
- *RQ2. Does* STUIOS *generate oracles that are effective in the presence of time uncertainty?* This research question aims to evaluate the effectiveness of *STUIOS* in terms of fault detection capability of the generated test suite when faults affecting timing requirements are characterized by time uncertainty.
- *RQ3. Can* STUIOS *negatively impact the efficiency of the testing process?* This question aims to evaluate whether *STUIOS* can be successfully adopted in an industrial context and what are the costs associated with its adoption.

5.4.1 RQ1. How does *STUIOS* compare with a simpler and less expensive alternative?

To respond to *RQ1*, we compared *STUIOS* oracles with oracles that do not include UIO sequences, which are the kind of oracles generated by traditional approaches relying on analyzing outputs sequences. There are no approaches that focus on deriving UIO sequences from specifications provided as TAs.

To perform the experiment, and avoid bias in the selection of the test cases, we considered the suite of 122 test cases used for the experiments described in Chapter 4 (Section 4.8). From the 122 test cases, we derived two test suites: (1) a test suite made of the 122 provided test cases, which include only simple oracles, and (2) a test suite containing the stochastic test cases generated by *STUIOS* by extending the 122 provided test cases with PUIO sequences and probabilistic information.

We compared the two test suites considering fault detection effectiveness. To this end we relied on mutation analysis. We derived the faulty versions of *BodySense* by means of specification mutation (i.e., by using mutation operators to derive software specifications that characterize the behaviour of a faulty program). In the literature, different mutation operators for TAs have been described (e.g., [Aboutrab et al., 2013, Aichernig et al., 2013]). For our experiments, we considered the same mutation operators adopted in the experiments described in Chapter 4, which are: Restricting Clock Conditions [Aboutrab et al., 2013], Widening Clock Conditions [Aboutrab et al., 2013], Shifting Clock Conditions [Aboutrab et al., 2013], and Change Target Location [Aichernig et al., 2013]. Applying the above mutation operators led to 323 mutated specifications for *BodySense*.

Manually modifying the implementation of *BodySense* to create 323 faulty versions that behave as the mutated specifications, and deploying the faulty versions on the dedicated hardware for purely experimental purposes, is far too expensive since it would require huge engineering effort. For this reason, we generated 323 programs that implement the mutated specifications of *BodySense*. Since these programs are generated for experimental purposes, they only implement the behavior under test related to timeliness and are therefore much simpler. We have used the 323 generated programs to evaluate the effectiveness of the stochastic test cases generated by *STUIOS*.

We executed each test case of the two test suites on each of the faulty software versions (mutants). In the case of *STUIOS*, each test case was executed multiple times on the software under test as required by *STUIOS*, and we kept track of passing and failing test cases. We measured fault detection effectiveness at the test suite level and report the percentage of faults discovered by the two test suites. Both test suites never report failures when executed against the valid version of the software.

The empirical results show that the test suite generated by *STUIOS* is more effective than the traditional test suite since it is able to discover 100% of the faults, while the test suite integrating simple traditional oracles discover only 91% of the faults. More precisely, the test suite integrating the simple traditional oracles does not detect all the transition faults, which are instead successfully discovered by the test suite generated by *STUIOS*.

5.4.2 RQ2. Does *STUIOS* generate oracles that are effective in the presence of time uncertainty?

To respond to *RQ2*, we evaluated the effectiveness of *STUIOS* in the presence of time uncertainty. More precisely we are interested in evaluating the effectiveness of the oracles generated by *STUIOS* (i.e., the capability of detecting failures), when failures are related to software timeliness (i.e., the capability of satisfying timing constraints), and the executed test cases lead to non-deterministic behaviour because of time uncertainty. To this end we evaluated the capability of *STUIOS* oracles in detecting failures that regard the timing requirements of *BodySense*. More specifically we focussed on failures that may affect the capability of *BodySense* of qualifying error conditions (e.g., the presence

of a temperature value out of range) within the given time ranges.

For our experiments, we considered both a valid implementation of the *BodySense* specifications and 44 faulty versions. The latter is a subset of the 323 faulty versions considered to address *RQ1*. We selected only mutated specifications derived by applying the mutation operators Restricting Clock Conditions and Widening Clock Conditions on the TA edges that capture the qualification time of each different error condition, thus mimicking faults that lead to error conditions being qualified with an erroneous timing. Applying the two mutation operators on the *BodySense* specifications lead to two different faulty versions for each TA regarding the detection of error conditions, each containing a single fault. Since *BodySense* specifications includes 22 TAs regarding error conditions, one for each different kind of error condition detected at runtime by *BodySense*, this is how we end up with 44 faulty versions of *BodySense*.

To perform our experiments, we did not consider all 122 test cases executed to answer RQ1 since many of these test cases stress deterministic behaviors of *BodySense* only. For example, the subset of test cases that cover the qualification of error conditions simply checks if the system qualifies error conditions within the maximum allowed time bound, and thus, are not affected by the non-deterministic behaviour caused by time uncertainty.

To address RQ2, we thus considered a test suite of 22 test cases that cover non-deterministic behaviours. The considered test cases check if *BodySense* is able to qualify errors within 3950 ms (i.e., the middle point in the allowed range) with the expected probability. Each test case verifies the timing constraints of a single error condition managed by *BodySense*.

We executed each test case against both the valid version and the two faulty versions for a total of 66 test case runs (please note that each run may imply repeated executions of the test case because of the probabilistic nature of *STUIOS*). Test cases executed against the valid version are expected to pass while test cases executed against faulty versions are expected to fail.

To respond to *RQ2*, we computed the precision and recall according to standard formulas. In our context, true positives correspond to test cases correctly failing against the faulty version, whereas false positives correspond to test cases unexpectedly failing against the correct version. True negatives match test cases not failing when executed against a faulty version.

Empirical results show that *STUIOS* provides perfect precision and recall (1.0). Please note that test cases with simple oracles simply cannot be executed in this context because they would fail every time the system does not qualify errors in 3950 ms, which happens in 50% of the passing executions since in the case of *BodySense* the qualification time follows a uniform distribution.

5.4.3 RQ3. Can *STUIOS* negatively impact the efficiency of the testing process?

STUIOS requires multiple executions of the same test case in order to determine the test outcome. This could clearly be an issue when test execution is manual because of the time spent by software engineers in running the test cases (though, for safety-critical software, such costs might be justified by increased fault detection capability).

In our context, like in many other embedded development environments, the repeated execution of test cases is not a problem. Software engineers are expected to set up the physical environment only once, at the beginning of the testing process, and as in many other industrial contexts, the execution of test suites is usually conducted automatically by test automation frameworks [Garousi and Mäntylä, 2016]. The efficiency of the testing process might thus be affected only when the number of test case executions is so high that such executions cannot be performed within available time. However, the number of test cases in such embedded systems, especially when using effective strategies such as the one implemented by TAUC, is usually not very large.

To respond to RQ3, we compared the number of test executions required to generate test results with *STUIOS* against those required by the simple test approach, i.e., 122 execution, one for each of the 122 test cases. In our empirical study, we determined that *STUIOS* requires 57 executions, on average, for each test case, and a minimum and maximum number of 35 and 100 executions. Although the number of test case executions is high, the execution time is acceptable in practice. Indeed, each test case takes on average 2.5 minutes to execute on the actual execution environment. This matches a total execution time of 290 hours which is a bit more than 10 days. Given that these test cases can be concurrently executed on several test environments and that acceptance testing usually lasts days, the fact that *STUIOS* significantly increases the number of test cases executions has limited practical consequences.

5.4.4 Threats to Validity

5.4.4.1 Internal threats

To limit the threats to the internal validity of the empirical evaluation, that is, a faulty implementation of our toolset that may lead to erroneous results, we have carefully tested our prototype implementation. In addition to this, we have manually inspected all the generated test suites. In particular we have manually verified that all the UIO sequences generated by our toolset are valid, and that the generated stochastic test cases contain the correct pair of test inputs and PUIO sequences. We did not verified if the probabilities calculated for PUIO sequences and test outputs are correct because we rely upon UPPAAL for this functionality, and we assume a correct implementation of the tool.

5.4.4.2 External threats

Threats to the external validity concern the generalisability of results. More precisely we observe two potential problems, the first is related to the representativeness of *BodySense*, the second concerns our choice of adopting a synthetic implementation of *BodySense* for our experiments.

BodySense is a control system implemented by means of a main control loop that poll data from the environment and then perform other operations. These kinds of control components are common in embedded and cyber-physical system, and we thus believe that many software systems might benefit from *STUIOS*. In addition to this, we believe that *BodySense* is a system with a typical complexity (in terms of size and number of models required to capture its timing properties) for software of that kind, and thus we expect that *STUIOS* can be successfully adopted to test other similar systems.

In our experiments, we have used synthetic implementations of *BodySense* that behave as indicated by a set of faulty models generated from the valid models of *BodySense*. Although these systems

are not real implementations of BodySense and do not integrate all the features of BodySense (e.g., they do not work on the LIN bus), they behave exactly as BodySense for the subset of features that we have considered for our experiment, more precisely they detect error conditions and signal them according to the given timing specifications.

5.5 Conclusions

In this chapter, we have proposed an approach named *STUIOS* that addresses the automated generation of effective test oracles for timeliness test cases in the presence of uncertainty.

STUIOS generates stochastic test cases that extend a set of test cases provided by software engineers. A stochastic test case generated by *STUIOS* includes the input-output sequences from the initial test cases, a probability value that indicates the expected probability of observing the output sequence generated by the test case, plus a *probabilistic UIO sequence*. A probabilistic UIO sequence captures the probability of observing the output of a UIO sequence, identified by *STUIOS*, which determines if the test case has terminated in the desired final state. Probability values capture the expected frequency of non-deterministic output sequences and thus allow *STUIOS* to deal with non-determinism. *STUIOS* then relies on executing multiple times each test case in such as way that it can employ hypothesis testing to determine whether a test case has passed (i.e., if it has generated the expected output with the expected frequency).

Results from an industrial case study in the automotive domain demonstrated that *STUIOS* is able to achieve higher fault detection effectiveness compared to the simpler test cases generated by TAUC (described in chapter 4). In practice, though *STUIOS* expectedly requires significant additional test execution time, this can be considered to have negligible practical consequences if case test execution is automated.

Chapter 6

Related Work

This chapter provides an overview of existing work related to the approaches researched and developed for this dissertation: automated generation of functional system test cases, automated testing of software timeliness, and definition of oracles for non-deterministic systems charachterized by time uncertainty.

6.1 Generation of Functional System Test Cases

Several approaches require that system requirements are given in UML behavioral models such as activity diagrams [Linzhang et al., 2004] [Nayak and Samanta, 2011], statecharts [Ryser and Glinz, 1999] [Bandyopadhyay and Ghosh, 2009], and sequence diagrams [Briand and Labiche, 2002] [Nebut et al., 2006]. Nebut et al. [Nebut et al., 2006] propose a use case driven test generation approach based on system sequence diagrams. Briand and Labiche [Briand and Labiche, 2002] use both activity and sequence diagrams to generate system test cases. While sequential dependencies between use cases are extracted from an activity diagram, sequences in a use case are derived from system sequence diagrams. In contrast, UMTG only requires use case specifications complemented by a domain model including OCL constraints, the latter being required even for techniques relying on behavioural models to generate test data and executable test cases.

There are approaches generating UML behavioral models from NL requirements [Yue et al., 2013] [Yue et al., 2010] [Frohlich and Link, 2000]. For example, Yue et al. [Yue et al., 2013] [Yue et al., 2011] generate UML state machines from RUCM use cases. One issue that prevents using such an approach for automated test generation is that software engineers have to correct and complete the complex, generated models, thus forcing engineers to deal with complex behavioural models, which we mean to avoid in our work. Frohlich and Link [Frohlich and Link, 2000] show how use cases can be systematically transformed into UML state charts, from which test cases are derived. The approach proposed by Frohlich and Link has two drawbacks: (i) generated test sequences have to be edited and (ii) test data have to be manually provided. In contrast, UMTG not only generates sequences of function calls but also generates test data for these functions.

Different approaches use NLP techniques to derive test cases from NL requirements. Unfortunately, most of these approaches require that generated test cases be manually augmented with missing data [Escalona et al., 2011]. In very recent work, developed concurrently with ours, Zang et

al. [Zhang et al., 2014] generate test cases from RUCM use cases by identifying sequences of use case steps that satisfy branch and loop coverage. Test cases cannot be executed automatically because they do not include concrete test data. In contrast, UMTG introduces some extensions in RUCM which, combined with the use of OCL constraints on a domain model, enable the generation of executable test cases with concrete test data. Another recent approach that generates test cases without test data from NL requirements is that of Sarmiento et al. [Sarmiento et al., 2014].

Carvalho et al. [Carvalho et al., 2013] generate executable test cases for reactive systems, from requirements written according to a restricted grammar and dictionary. There are two main limitations: the underlying dictionary may change from project to project, while a restricted grammar may not be suitable to express the requirements of different kinds of systems. UMTG does not impose a restricted dictionary for use cases, but simply relies on a generic and restricted use case format that can be used to express use cases of different kinds of systems.

Similar to our approach, Archetest [Kaplan et al., 2008] requires a domain model and a use case specification with invariants and processable guard and post conditions. Unfortunately, Archetest can generate only test data without test case scenarios.

6.2 Automated testing of software timeliness

Techniques that support testing of software timeliness include model-based approaches, and approaches based on meta-heuristic search.

Model-based approaches for testing software timeliness often rely upon timed automata [Hessel et al., 2008]. Most test generation approaches rely upon coverage strategies adapted from traditional finite state machine testing [Chow, 1978, Cardell-Oliver and Glover, 1998]. Other approaches adopt model checkers for test generation. In these cases, the model is typically annotated with auxiliary variables or automata that enable the formulation of the testing purpose or the coverage criterion as a reachability problem [Hessel et al., 2008, Hessel et al., 2004]. Finally, some approaches rely upon coverage criteria explicitly defined for guard conditions over clock variables [Aboutrab et al., 2013, En-Nouaary and Hamou-Lhadj, 2008]. The main limitation of these approaches is that they require complete models, i.e. models that contain all the information required to identify the inputs to be sent to the system in order to trigger state transitions. Without complete models, model-based approaches can be used to generate abstract test cases only, which then need to be concretized through test adaptation, or test transformation approaches [Utting and Legeard, 2006]. TAUC instead requires minimal modelling of the timing properties and no additional adaptation or transformation layers. It further relies upon use case specifications, which are commonly used in many domains for communication purposes, to generate the test inputs that lead to targeted state transitions.

Other approaches rely upon other formalisms such as UML Statecharts [Mücke and Huhn, 2004], Extended Finite State Machines [Zheng et al., 2008], and Attribute Event Grammars [Auguston et al., 2005], but they share the same practical limitations mentioned above with approaches based on timed automata.

Metaheuristic search has been successfully applied to testing deadline misses and computing worst-case execution time. Di Alesio et al. combine genetic algorithms and constraint programming

to identify scenarios in which the tasks execution time is close or breaks the deadline [Di Alesio et al., 2015]. TAUC has a different goal in a different context, i.e. the system testing of timing requirements, expressed as timed automata, based on information available in use case specifications.

Iqbal et al. [Iqbal et al., 2012] use search-based algorithms to stress test real-time software by repeatedly executing the software while simulating the environment. Their goal, which is quite different from ours, is to reach critical error states. Another main difference is that TAUC does not require the execution of test cases, thus being more suitable when test cases execution is expensive or the environment cannot be fully simulated.

A few approaches use genetic algorithms to generate test cases specifically targeting timing properties specific to interrupt handlers [Regehr, 2005, Yu et al., 2014]. In contrast, TAUC is generally applicable to all systems whose functionality is captured as use case specifications.

6.3 Definition of oracles for systems charachterized by time uncertainty

Related work includes techniques for the generation of test cases and oracles from state-based formalisms, e.g., finite state machines (FSMs), TAs, and Extended FSMs (EFSMs), as well as approaches for testing real-time systems.

Most of the related work for the generation of test cases and oracles based on finite-state specifications deals with FSMs, not TAs. Two surveys are relevant, one focusing on FSM-based testing approaches [Lee and Yannakakis, 1996] and a more recent one also including experimental comparisons [Dorofeeva et al., 2010]. Other approaches focus on EFSMs [Duale and Uyar, 2004, Robinson-Mallett et al., 2006], but require deterministic EFSMs and thus cannot be applied in the context of this paper, as discussed in Section 5.3.3.

On the other hand, several approaches working with TAs target the generation of test cases and do not address the problem of generating precise test oracles. For example, Springintveld *et al.* introduced a technique for translating TAs into grid automata to enable generation of test sequences by means of traditional coverage-based algorithms for FSM testing [Springintveld et al., 2001].

AbouTrab *et al.* [Trab et al., 2010, Aboutrab et al., 2013] use region automata to generate test traces that satisfy the transition coverage criterion. EnNouaary et al. [En-Nouaary, 2008] rely upon sampling to convert a TA into a traditional FSM and then generate test cases.

Few techniques focussing on testing real-time systems in the presence of time uncertainty exist [David et al., 2012, Garousi, 2008, David et al., 2009, Hierons and Merayo, 2009]. The main difference with *STUIOS* is that these approaches focus on test case generation and do not deal with the problem of generating test oracles for existing test suites, as described next.

In [David et al., 2012], David *et al.* present a survey of approaches for testing real-time systems under partial observability and uncertainty in the presence of specifications given as TAs and Timed Game Automata (i.e., TAs with the set of edges partitioned into controllable ones and uncontrollable ones). Garousi [Garousi, 2008] instead proposes a stress testing methodology to detect network-

traffic-related faults in real-time systems based on the design UML models in the presence of time uncertainty (i.e., when the timing information of messages is imprecise or unpredictable). Differently from other approaches, the methodology is not based on formal specifications of the system but on the *barrier* scheduling heuristic, taken from the operating systems literature.

The two approaches closest to *STUIOS* are that of David et al. [David et al., 2009] and Hierons et al. [Hierons and Merayo, 2009]. David *et al.* [David et al., 2009] generate test cases under "partial observability". They model a given SUT using Timed Game Automata (TGA) with internal actions, uncontrollable outputs and timing uncertainty of outputs and generate test cases by relying upon a stochastic extension of UPPAAL. Test generation is driven by test requirements specified in Computation Tree Logic (CTL) formulas. The main difference with *STUIOS* is that David *et al.* perform test generation based on CTL objectives — thus preventing engineers from reusing or extending test suites derived with different criteria — while *STUIOS* focus on the generation of oracles for existing test suites.

The test approach proposed by Hierons *et al.* [Hierons and Merayo, 2009] relies on mutation testing and probability estimation to build test suites that verify if a software conforms to specifications given as probabilistic FSMs. The approach derives test cases that aim to spot specific implementation faults. To this end, the approach identifies sequences of test inputs that can determine if a mutated specification, which models a fault, differs from the original one. Like in the case of David *et al.* the approach proposed by Hierons *et al.* cannot be applied to derive oracles for existing test suites. Furthermore, its effectiveness has not been evaluated and it cannot handle specifications given as a network of TAs (typical in industrial systems).

Chapter 7

Conclusions and Future Work

This chapter is organised as follows. Section 7.1 summarises the contributions of this dissertation. Section 7.2 discusses potential future work.

7.1 Summary

In this dissertation, we address the problem of automatically generating system test cases that verify the compliance of an embedded software system with its functional and timing requirements while minimizing the effort needed to produce the artefacts (e.g., models) required to enable automated test case generation. We focus on functional and timing requirements because these are usually of crucial importance for ensuring proper system behavior and safety. To make our approach practical we have focused on solutions targeting common embedded development contexts where functional and timing requirements are expressed by means of use case specifications and timed automata, respectively. The techniques developed in the context of this thesis have been validated against an industrial case study, *BodySense*, a representative embedded system developed by IEE, a company located in Luxembourg.

In this dissertation, we presented a set of coherent guidelines for the analysis and definition of functional and timing requirements, followed by three techniques that enable the automated generation of functional and timeliness test cases. The guidelines for requirement analysis enable engineers to adopt the test automation techniques presented in this dissertation. Some of the proposed guidelines, for example, the adoption of a template for the definition of use case specifications, have been adopted by our industrial partner, as they were convinced of its benefits.

The first test automation technique presented in this dissertation, UMTG, integrates a natural language processing solution that enables the automated generation of test cases from use case specifications. The additional modelling effort required by UMTG to enable testing consists of the definition of OCL constraints to clarify the conditional statements in use case specifications, and the pre- and post-conditions of use case flows. Our experience with IEE has shown that the required OCL constraints helped engineers make software specifications more precise by resolving ambiguities. In addition to this, our empirical evaluation has shown that the generated test cases cover important use case scenarios not covered by the current test suites developed by engineers based on domain expertise.

The second technique presented in the dissertation, TAUC, automatically generates executable

test cases for testing software timeliness from high-level specifications of timing requirements given as timed automata. To this end, TAUC exploits the test cases automatically generated by UMTG. More specifically it identifies dependencies between the events consumed in the timed automata and the use case specifications in order to identify sequences of functional test cases that trigger the events required to cover timeliness requirements. In addition, the technique integrates meta-heuristic search to generate test cases that stress timing requirements by maximizing test suite diversity. Our empirical evaluation has shown that the test suite generated by TAUC can detect more faults than the manual test suite defined by IEE software engineers.

The third technique presented in this dissertation, *STUIOS*, automatically generates oracles for testing non-deterministic systems where the source of non-determinism is time uncertainty, a typical occurrence in embedded systems. *STUIOS* integrates statistical model checking and statistical testing in order to determine if the output of a test case including a unique input output sequence occurs with the expected frequency. The empirical results obtained so far have shown that *STUIOS* enables the detection of more faults than traditional test cases including simpler oracles.

7.2 Future Work

In the future, we aim to both consolidate the techniques developed in the context of this dissertation and define new techniques that build upon them to achieve full automation of test case generation.

As already mentioned in the previous chapters some of the guidelines proposed in this dissertation have been already integrated in the development process of IEE. We have delivered two workshops about UMTG and the supporting modelling guidelines that enabled software engineers at IEE to adopt the proposed techniques. As a consequence, we will be able to perform additional studies about the effectiveness and fault detection capability of UMTG. This will in turn also enable additional evaluations of TAUC. TAUC has not been adopted by IEE yet, but it is our plan to deliver workshops to support engineers in the adoption of the technique.

Among all the models required by the techniques proposed in this dissertation the only inputs required for testing purposes are the OCL constraints. The automatic generation of OCL constraints will enable the fully automated generation of system test cases from functional requirements. To achieve this goal we will focus on the development of techniques for the automated generation of the OCL constraints required by UMTG from the use case specifications written in natural language. Existing approaches that generate executable test cases from requirements written in natural language either require that software specifications are written according to very restricted domain specific languages [de Figueiredo et al., 2006, Carvalho et al., 2013], or they require the same additional modelling effort required by UMTG [Kaplan et al., 2008]. We aim to exploit recent advances in semantic analysis based on natural language processing (e.g., the ones that enable arithmetic word problem solving [Punyakanok et al., 2008, Roy and Roth, 2016]) to achieve this goal without imposing too restrictive a language for defining use case specifications.

List of Papers

Published papers included in this dissertation:

- Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. "Automatic Generation of System Test Cases from Use Case Specifications." In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 385-396. ACM, 2015.
- Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. "UMTG: a toolset to automatically generate system test cases from use case specifications." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 942-945. ACM, 2015., pp. 942-945. ACM, 2015.
- Chunhui Wang, Fabrizio Pastore, and Lionel Briand. "System Testing of Timing Requirements Based on Use Cases and Timed Automata." In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pp. 299-309. IEEE, 2017.

Unpublished papers included in this dissertation:

- Chunhui Wang, Fabrizio Pastore, and Lionel Briand. "Oracles for Testing So ware Timeliness with Uncertainty." Under review.

Bibliography

[Aboutrab et al., 2013] Aboutrab, M. S., Brockway, M., Counsell, S., and Hierons, R. M. (2013). Testing real-time embedded systems using timed automata based approaches. *J. Syst. Softw.*, 86(5):1209–1223.

[Aichernig et al., 2013] Aichernig, B. K., Lorber, F., and Ničković, D. (2013). *Time for Mutants — Model-Based Mutation Testing with Timed Automata*, pages 20–38. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Ali et al., 2013] Ali, S., Zohaib Iqbal, M., Arcuri, A., and Briand, L. (2013). Generating test data from ocl constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10):1376–1402.

[Alur and Dill, 1994] Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235.

[Anastasakis et al., 2007] Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2007). Uml2alloy: A challenging model transformation. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, MODELS'07, pages 436–450, Berlin, Heidelberg. Springer-Verlag.

[Auguston et al., 2005] Auguston, M., Michael, J. B., and Shing, M.-T. (2005). Environment behavior models for scenario generation and testing automation. In *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, A-MOST '05, pages 1–6, New York, NY, USA. ACM.

[Bandyopadhyay and Ghosh, 2009] Bandyopadhyay, A. and Ghosh, S. (2009). Test input generation using uml sequence and state machines models. In *ICST'09*, pages 121–130.

[Bengtsson et al., 1995] Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P., and Yi, W. (1995). Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag.

[Bittner, 2002] Bittner, K. (2002). *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Briand and Labiche, 2002] Briand, L. C. and Labiche, Y. (2002). A uml-based approach to system testing. *SoSyM*, 1(1):10–42.

[Cardell-Oliver and Glover, 1998] Cardell-Oliver, R. and Glover, T. (1998). A practical and complete algorithm for testing real-time systems. In Ravn, A. and Rischel, H., editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 251–261. Springer Berlin Heidelberg.

[Carvalho et al., 2013] Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., and Blackburn, M. (2013). Test case generation from natural language requirements based on SCR specifications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*, pages 1217–1222.

[Chow, 1978] Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187.

[Cockburn, 2000] Cockburn, A. (2000). *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

[Cunningham et al., 2002] Cunningham, H., Maynard, D., Bontcheva, K., and Tablan, V. (2002). GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *ACL'02*.

[David et al., 2012] David, A., Larsen, K. G., Li, S., Mikucionis, M., and Nielsen, B. (2012). *Testing Real-Time Systems under Uncertainty*, pages 352–371. Springer Berlin Heidelberg, Berlin, Heidelberg.

[David et al., 2009] David, A., Larsen, K. G., Li, S., and Nielsen, B. (2009). Timed testing under partial observability. In *2009 International Conference on Software Testing Verification and Validation*, pages 61–70.

[David et al., 2002] David, A., Möller, M. O., and Yi, W. (2002). *Formal Verification of UML Statecharts with Real-Time Extensions*, pages 218–232. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Davis and Burns, 2011] Davis, R. I. and Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44.

[de Figueiredo et al., 2006] de Figueiredo, A. L. L., Andrade, W. L., and Machado, P. D. L. (2006). Generating interaction test cases for mobile phone systems from use case specifications. *SIGSOFT Software Engineering Notes*, 31(6):1–10.

[Derderian et al., 2006] Derderian, K., Hierons, R. M., Harman, M., and Guo, Q. (2006). Automated unique input output sequence generation for conformance testing of fsms. *Comput. J.*, 49(3):331–344.

[Di Alesio et al., 2015] Di Alesio, S., Nejati, S., Briand, L., and Gotlieb, A. (2015). Combining genetic algorithms and constraint programming to support stress testing of task deadlines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*.

[Dorofeeva et al., 2010] Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A. R., and Yevtushenko, N. (2010). Fsm-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12):1286 – 1297.

[Duale and Uyar, 2004] Duale, A. Y. and Uyar, M. U. (2004). A method enabling feasible conformance test sequence generation for efsm models. *IEEE Transactions on Computers*, 53(5):614–627.

[Eclipse, 2017] Eclipse (2017). Eclipse IDE. http://www.eclipse.org.

[Eclipse OCL, 2017] Eclipse OCL (2017). Eclipse OCL. http://www.eclipse.org/modeling/.

[En-Nouaary, 2008] En-Nouaary, A. (2008). A scalable method for testing real-time systems. *Software Quality Journal*, 16(1):3–22.

[En-Nouaary et al., 2002] En-Nouaary, A., Dssouli, R., and Khendek, F. (2002). Timed wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.*, 28(11):1023–1038.

[En-Nouaary and Hamou-Lhadj, 2008] En-Nouaary, A. and Hamou-Lhadj, A. (2008). A boundary checking technique for testing real-time systems modeled as timed input output automata (short paper). In *2008 The Eighth International Conference on Quality Software*, pages 209–215.

[Escalona et al., 2011] Escalona, M. J., Gutierrez, J. J., Mejías, M., Aragón, G., Ramos, I., Torres, J., and Domínguez, F. J. (2011). An overview on test generation from functional requirements. *Journal of System and Software*, 84(8):1379–1393.

[Frohlich and Link, 2000] Frohlich, P. and Link, J. (2000). Automated test case generation from dynamic models. In *ECOOP'00*, pages 472–491.

[Garousi, 2008] Garousi, V. (2008). Traffic-aware stress testing of distributed real-time systems based on uml models in the presence of time uncertainty. In *International Conference on Software Testing, Verification, and Validation*, pages 92–101.

[Garousi and Mäntylä, 2016] Garousi, V. and Mäntylä, M. V. (2016). When and what to automate in software testing? a multi-vocal literature review. *Information and Software Technology*, 76:92 – 117.

[Harrold et al., 1998] Harrold, M. J., Rothermel, G., and Sinha, S. (1998). Computation of interprocedural control dependence. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 11–20, New York, NY, USA. ACM.

[Hemmati et al., 2013] Hemmati, H., Arcuri, A., and Briand, L. (2013). Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):6.

[Hérault et al., 2004] Hérault, T., Lassaigne, R., Magniette, F., and Peyronnet, S. (2004). *Approximate Probabilistic Model Checking*, pages 73–84. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Hessel et al., 2008] Hessel, A., Larsen, K., Mikucionis, M., Nielsen, B., Pettersson, P., and Skou, A. (2008). Testing real-time systems using uppaal. In Hierons, R., Bowen, J., and Harman, M., editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer Berlin Heidelberg.

[Hessel et al., 2004] Hessel, A., Larsen, K., Nielsen, B., Pettersson, P., and Skou, A. (2004). Time-optimal real-time test case generation using uppaal. In Petrenko, A. and Ulrich, A., editors, *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 114–130. Springer Berlin Heidelberg.

[Hierons and Merayo, 2009] Hierons, R. M. and Merayo, M. G. (2009). Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software*, 82(11):1804 – 1818. SI: {TAIC} {PART} 2007 and {MUTATION} 2007.

[IBM Doors, 2017] IBM Doors (2017). IBM Doors. http://www-03.ibm.com/software/products/en/ratidoor.

[IBM Rhapsody, 2017] IBM Rhapsody (2017). IBM Rhapsody. http://www-03.ibm.com/software/products/en/ratirhapfami.

[Iqbal et al., 2012] Iqbal, M. Z., Arcuri, A., and Briand, L. (2012). Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 199–209, New York, NY, USA. ACM.

[ISO, 2011] ISO (2011). ISO-26262: Road vehicles – functional safety.

[Kaplan et al., 2008] Kaplan, M., Klinger, T., Paradkar, A. M., Sinha, A., Williams, C., and Yilmaz, C. (2008). Less is more: A minimalistic approach to UML model-based conformance test generation. In *1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 82–91.

[Krichen and Tripakis, 2004] Krichen, M. and Tripakis, S. (2004). *Black-Box Conformance Testing for Real-Time Systems*, pages 109–126. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Larman, 2002] Larman, C. (2002). *Applying UML and Patterns:An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall Professional.

[Larsen et al., 2005] Larsen, K. G., Mikucionis, M., Nielsen, B., and Skou, A. (2005). Testing real-time embedded software using uppaal-tron: An industrial case study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 299–306, New York, NY, USA. ACM.

[Lee and Yannakakis, 1996] Lee, D. and Yannakakis, M. (1996). Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

[Linzhang et al., 2004] Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., and Guoliang, Z. (2004). Generating test cases from UML activity diagram based on gray-box method. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*.

[Micrium Embedded Software, 2016] Micrium Embedded Software (2016). Micro c/os. https://www.micrium.com.

[Mücke and Huhn, 2004] Mücke, T. and Huhn, M. (2004). Generation of optimized testsuites for uml statecharts with time. In Groz, R. and Hierons, R., editors, *Testing of Communicating Systems*, volume 2978 of *Lecture Notes in Computer Science*, pages 128–143. Springer Berlin Heidelberg.

[Nayak and Samanta, 2011] Nayak, A. and Samanta, D. (2011). Synthesis of test scenarios using uml activity diagrams. *SoSyM*, 10:63–89.

[Nebut et al., 2006] Nebut, C., Fleurey, F., Le-Traon, Y., and Jezequel, J.-M. (2006). Automatic test generation: a use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155.

[OMG, 2004] OMG (2004). The Object Constraint Language (OCL). http://www.omg.org/spec/OCL/.

[Patrick et al., 2016] Patrick, M., Craig, A. P., Cunniffe, N. J., Parry, M., and Gilligan, C. A. (2016). Testing stochastic software using pseudo-oracles. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 235–246, New York, NY, USA. ACM.

[Phalp et al., 2007] Phalp, K. T., Vincent, J., and Cox, K. (2007). Improving the quality of use case descriptions: empirical assessment of writing guidelines. *Software Quality Journal*, 15(4):383–399.

[Punyakanok et al., 2008] Punyakanok, V., Roth, D., and Yih, W. (2008). The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics*, 34(2).

[R Core Team, 2013] R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

[Ramalingom et al., 1996] Ramalingom, T., Thulasiraman, K., and Das, A. (1996). Context independent unique sequences generation for protocol testing. In *INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 3, pages 1141–1148 vol.3.

[Regehr, 2005] Regehr, J. (2005). Random testing of interrupt-driven software. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 290–298, New York, NY, USA. ACM.

[Robinson-Mallett et al., 2006] Robinson-Mallett, C., Liggesmeyer, P., Mücke, T., and Goltz, U. (2006). Extended state identification and verification using a model checker. *Information and Software Technology*, 48(10):981–992.

[Roy and Roth, 2016] Roy, S. and Roth, D. (2016). Unit dependency graph and its application to arithmetic word problem solving. *CoRR*, abs/1612.00969.

[Ryser and Glinz, 1999] Ryser, J. and Glinz, M. (1999). A scenario-based approach to validating and testing software systems using statecharts. In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99)*.

[Sarmiento et al., 2014] Sarmiento, E., Leite, J. C. S. d. P., and Almentero, E. (2014). C&l: Generating model based test cases from natural language requirements descriptions. In *RET'14*, pages 32–38.

[Springintveld et al., 2001] Springintveld, J., Vaandrager, F., and D'Argenio, P. R. (2001). Testing timed automata. *Theor. Comput. Sci.*, 254(1-2):225–257.

[Trab et al., 2010] Trab, M. S. A., Alrouh, B., Counsell, S., Hierons, R. M., and Ghinea, G. (2010). *A Multi-criteria Decision Making Framework for Real Time Model-Based Testing*, pages 194–197. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Utting and Legeard, 2006] Utting, M. and Legeard, B. (2006). *Practical Model Based Testing*, chapter 8. Morgan Kaufmann Publishers.

[Vector, 2017] Vector (2017). CANoe Testing Tool by Vector. https://vector.com/.

[Ševčíková et al., 2006] Ševčíková, H., Borning, A., Socha, D., and Bleek, W.-G. (2006). Automated testing of stochastic systems: A statistically grounded approach. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 215–224, New York, NY, USA. ACM.

[Wang et al., 2017] Wang, C., Pastore, F., and Briand, L. (2017). System testing of timing requirements based on use cases and timed automata. In *ICST*.

[Wang et al., 2015] Wang, C., Pastore, F., Goknil, A., Briand, L., and Iqbal, Z. (2015). Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 385–396, New York, NY, USA. ACM.

[Yoo, 2010] Yoo, S. (2010). Metamorphic testing of stochastic optimisation. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 00:192–201.

[Yu et al., 2014] Yu, T., Srisa-an, W., Cohen, M. B., and Rothermel, G. (2014). Simlatte: A framework to support testing for worst-case interrupt latencies in embedded software. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST'14)*, pages 313–322. IEEE.

[Yue et al., 2011] Yue, T., Ali, S., and Briand, L. (2011). Automated transition from use cases to uml state machines to support state-based testing. In *ECMFA'11*, pages 115–131.

[Yue et al., 2010] Yue, T., Briand, L., and Labiche, Y. (2010). An automated approach to transform use cases into activity diagrams. In *ECMFA'10*, pages 337–353.

[Yue et al., 2013] Yue, T., Briand, L. C., and Labiche, Y. (2013). Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Transactions on Software Engineering and Methodology*, 22(1).

[Zhang et al., 2014] Zhang, M., Yue, T., Ali, S., Zhang, H., and Wu, J. (2014). A systematic approach to automatically derive test cases from use cases specified in restricted natural languages. In *SAM'14*, pages 142–157.

[Zheng et al., 2008] Zheng, M., Alagar, V., and Ormandjieva, O. (2008). Automated generation of test suites from formal specifications of real-time reactive systems. *Journal of Systems and Software*, 81(2):286–304.

[Zou et al., 2009] Zou, G. Y., Huang, W., and Zhang, X. (2009). A note on confidence interval estimation for a linear function of binomial proportions. *Computational Statistics and Data Analysis*, 53(4):1080 – 1085.

Appendix A

Tool Suite Description

A tool suite was developed to automate the approaches that we propose in this dissertation. This chapter provides an overview of the architecture of implemented tools.

A.1 UMTG Toolset

We have implemented UMTG as a Java toolset that is integrated with two widely used requirement and design tools that are adopted by our industry partner: IBM Doors [IBM Doors, 2017] and IBM Rhapsody [IBM Rhapsody, 2017] (UMTG works with the version of Rhapsody integrated with the Eclipse development environment [Eclipse, 2017]). Figure A.1 shows the UMTG architecture.

The main components of UMTG are two plug-ins that extend Doors and Rhapsody. These plug-ins provide the user interface of UMTG and orchestrate the other components of UMTG that implement the steps in Figure 3.1 (see Figure A.1 where the black circles denote the steps).

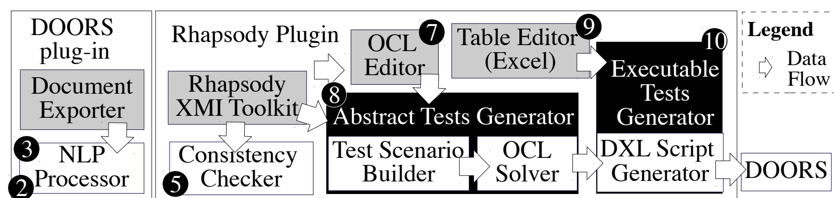


Figure A.1. UMTG architecture (grey boxes show third party components, black boxes UMTG components with nested components)

The UMTG Doors plug-in provides the menu buttons that activate the UMTG steps related to the elicitation of use cases while the UMTG Rhapsody plug-in provides the menu buttons that activate the steps related to test generation. Figure A.2 shows the contextual menus provided by the UMTG plug-ins where black circles denote the corresponding steps in Figure 3.1.

The UMTG Rhapsody plug-in also provides the interface to visualize and edit the artefacts produced by UMTG. UMTG takes advantage of the plug-in architecture of Eclipse to reuse functionality provided by third party plug-ins. In particular, UMTG relies upon the following user interfaces: the *Eclipse Web Browser* to visualize the domain entities missing from the domain model, the *OCL editor* provided by the Eclipse OCL plug-in [Eclipse OCL, 2017] to support software engineers in editing

the list of OCL constraints, the *Eclipse Text Editor* to visualize abstract test cases, the default *Eclipse Table Editor* (e.g. Microsoft Excel) to edit mapping tables, and the *Eclipse Project Explorer* to list the UMTG artefacts.

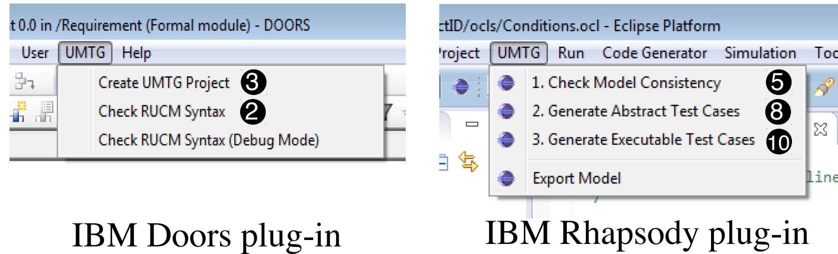


Figure A.2. Menus provided by the UMTG plug-ins

The other components in Figure A.1 implement the steps of the UMTG workflow automated by the UMTG toolset, i.e. Steps 2, 3, 5, 8, and 10 in Figure 3.1. The *NLP Processor* implements Steps 2 and 3. It is based on the GATE workbench [Cunningham et al., 2002], an open source NLP framework. To load use cases from IBM Doors, UMTG uses the *Doors Document Exporter*, an API that exports Doors content as text files.

The UMTG Rhapsody plug-in implements Steps 5, 8, and 10. The UMTG Rhapsody plug-in relies upon a third-party application, the *Rhapsody XMI Toolkit*, to export the domain model in the XMI format. This is necessary since Rhapsody saves models into its own proprietary format. The *Consistency Checker* and the *Abstract Tests Generator* implement steps 5 and 8. The *OCL Solver* is the constraint solver described in [Ali et al., 2013].

The *Executable Test Generator* implements Step 10, it takes the mapping table and the abstract test cases as input, and generates the executable test cases as output. The *Executable Test Generator* exploits the Door eXtension Language (DXL) to load the generated test cases into Doors. The DXL functionality is also used to automatically generate the traceability links between use case specifications and the generated test cases. UMTG adds to each generated test case a set of traceability links indicating the flow of the use case specifications covered by the test cases. Furthermore, for each use case flow covered by a test case, it generates traceability links indicating the test cases covering it.

ID	New Test Description	Input Values	Expected Value
TCC9856 (A)	<INPUT> BodySenseSensor.CableShieldIntegrityStatus = DomainModel::HWStatus::NotOK		
TCC9857 (B)	Set AMGB	Chanel = {RELAY_CABLE_SHIELD_INTEGRITY} Status = {ERROR}	
TCC9858	<CHECK> CableShieldIntegrityErrorIsDetected		
TCC9859 (C)	LIN Publish	MsgID = 2Dh	D1 = 52h D2 = 01h D3 = {ERR_CODE_(INTEGRITY}

Figure A.3. Example of an Abstract (left) and a corresponding Executable (right) Test Case Generated by UMTG

A.1.1 Test Cases Generation with The UMTG toolset

This section overviews a usage scenario of the UMTG toolset, with the aid of pictures of the implemented tool.

UMTG automatically checks the RUCM syntax. Syntax checking is required to ensure the proper functioning of the automated steps implemented by UMTG. Figure A.4 shows the window that indicates the presence of syntax errors. By clicking on each error the user is directly pointed to the erroneous step in IBM Doors.

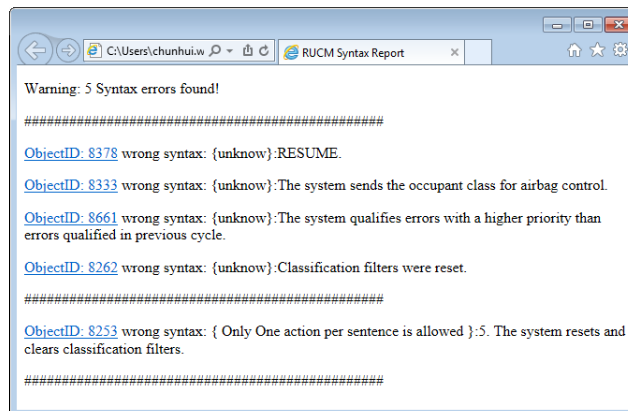


Figure A.4. RCUM Syntax Report

After verifying use cases syntax engineers can switch to Eclipse/Rhapsody to edit the domain model and very its consistency with the use case specifications (Figure 3.5).

The UMTG functionality for checking model consistency in UMTG (Figure A.2, label 5) shows the results of the consistency check in a popup window (Figure A.5). The result is shown in Figure A.5 indicates that the entity *ClassificationFilter* is missing and might be added by software engineers to the domain model.

Figure A.5. Consistency Check Report

The constraints to be provided by software engineers are then listed within Eclipse/Rhapsody in a file named *Conditions.ocl* (Figure A.6).

```
import 'file:/C:/Users/chunhui.wang/workspace/Demo/models/DomainModel.sbs.xmi'
/*
 *Uncomment the following line and replace the <Package Name>.
 */
package DomainModel

context BodySenseSystem

inv IsNotDiscarded: BodySenseSystem.allInstances()->forAll(b|b.discardFlag = false)

inv CalibrationIsDone: BodySenseSystem.allInstances()->forAll(b|b.calibrationDone = true)

inv NVMIsAccessible: BodySenseSystem.allInstances()->forAll(b|b.itsNVM.isAccessible = true)

inv BuildCheckIsDone: BodySenseSystem.allInstances()->forAll(b|b.buildCheckStatus <> BuildCheckStatus::NotRun)
```

Figure A.6. List of OCL Constraints

Abstract test cases are generated in the *ATS* folder of the Rhapsody workspace. Abstract test cases contain an header and a body (see Figure A.3).

UMTG generates two sets of test cases: abstract test cases and executable test cases (see Figure A.3).

Abstract test cases include test inputs and oracles expressed in terms of domain model entities. The left part of Figure A.3 shows an example of an abstract test case. Abstract test cases are saved in text files under the Eclipse/Rhapsody workspace. Each abstract test case includes a header section that depicts the steps of the scenario under test, and a body section with a list of input operations and oracles. The header lines in Figure A.3 show that the test case covers a scenario in which the condition *the cable shield integrity is valid* is false (header lines begin with the symbol #). One of the test inputs in Figure A.3 is an assignment of the value *NotOK* to *CableShieldIntegrityStatus* (this is the value that falsifies the condition above).

Step 9. A mapping table is provided by software engineers (Figure A.9).

Figure A.7. Mapping table

Step 10. Test cases with traceability links are directly generated in Doors (Figure A.8).

Figure A.8. Traceability Links Generated for a Test Case.

The executable test cases generated by UMTG extend the abstract test cases by including calls to the test driver functions that need to be invoked to execute the system.

The right part of Figure A.3 shows a portion of an executable test case generated by UMTG. The generated test case includes lines with abstract test inputs that are included to provide high-level operation descriptions. These lines are followed by the test driver functions with the concrete inputs to be used to test the system. Label A in Figure A.3 points to the high-level operation description that indicates that the test case must set the *CableShieldIntegrityStatus* to the ‘failed’ status. Label B points to the corresponding test driver function, i.e., *Set AMGB*. *Set AMGB* is used to simulate an input signal coming from a sensor (in this case it sends an error status on the channel of the *shield integrity* sensor). The generated test case also contains test oracles. Label C shows an oracle implemented by means of an invocation of function *Lin Publish*, which is used to check the signals sent on a channel.

Executable test cases are generated by means of a mapping table that is provided by software engineers. Figure A.9 shows a portion of the mapping table used for *BodySense*. To generate calls to driver functions, UMTG parses each line in the abstract test cases according to the mapping table. The mapping table is made of five columns. The first two columns provide operation names and regular expressions that match an input in the abstract test case. The last three columns provide the driver function calls and parameters that should be added to the executable test case when an abstract input matches the regular expression.

1	<INPUT>	BodySenseSensor.CableShieldIntegritySt atus = DomainModel::HWStatus::NotOK	Set AMGB	Channel = {RELAY_CABLE_SHIELD_INTEGRITY} Status = {ERROR}	
2	<CHECK>	CableShieldIntegrityErrorIsDetected	LIN Publish	MsgID=2Dh	D1=52h D2=01h D3={ERR_C

Figure A.9. Portion of a Mapping Table

A.2 TAUC Toolset

TAUC has been implemented as a set of Java programs integrated with both the UPPAAL model checker and UMTG. The programs belonging to TAUC have been wrapped into a set of executable executable jar files. In the following, we describe the developed programs according to their role, i.e. if they implement the TAUC technique and thus can be used to perform test generation, or if they are support tool that we adopted to perform the empirical evaluation.

A.2.1 Test Generation Tools

Two programs belong to this category *ttmGen* and *taucGen*. The program *ttmGen* processes two inputs: (1) a network of UPPAAL timed automata including timing requirements automata and environment automata, and (2) the functional test cases generated by UMTG. *ttmGen* generates as output a *Timeliness Test Model* in the UPPAAL TA format. *ttmGen* follows the process described in Chapter 4.5. First, *ttmGen* generates *scenario automata* from the UMTG functional test cases, then it identifies the dependencies between timing requirement automata and functional scenarios following the rule defined in Chapter 4.4 . Recall that the dependencies between timing requirement automata and functional scenarios are of two kinds: (1) outputs produced by functional scenarios may fire state transitions, (2) certain functional scenarios can be executed if and only if the system has reached

specific states, i.e. specific state invariants are true. Finally *ttmGen* generates the augmented timing requirement automata by augmenting the dependency information with the time timing requirement automata.

The program *taucGen* instead is the TAUC test case generator, and it is used to generate timeliness test suite after the *Timeliness Test Model* has been generated. *taucGen* takes as input a *Timeliness Test Model*. Figure A.10 shows the list of arguments accepted by *taucGen*.

```
~$ java -jar taucGen.jar
usage: taucGen.jar
 -i <InputModels>         File/Folder contains Timeliness Model.
 -d <DomainModel>         Domain Model File.
 -o <PathForTestSuite>    Specify the path to save test suite.
 -t <PathForTrace>        Specify the path to save traces.
 -m <NumberOfIteration>   Specify number of Iteration.
 -n <SizeOfTestSuite>     Size of generated test suite.
 -r                       Random test generation strategy instead TAUC.
 -p                       Print test generation statistics.
```

Figure A.10. Description of taucGen.jar

A.2.2 Empirical Evaluation Tools

In chapter 4.8 we have shown that we evaluated the effectiveness of TAUC against a set of modified timed automata that simulate the presence of faults in the system. To this end, we implemented three programs, *muTA*, *vmodels*, and *verifier*. The first generates mutated timeliness models, the second translates the test cases generated by TAUC into timed automata, the third simulates the execution of test cases against these models.

The program *muTA* automatically generates modified Timeliness test model by relying upon a subset of the mutation operators suggested in [Aboutrab et al., 2013, Aichernig et al., 2013] which may impact timing requirements. This enables the generation of mutants that represent faulty implementations of the system. Figure A.11 shows a description the program arguments of *muTA*.

```
~$ java -jar muta.jar
usage: muta.jar
 -f <ModelFile>       Timeliness model file to mutate.
 -m <MutaOps>         List of mutation operators.
 -o <PathForMutants>  Specify the path to save generated mutants.
 -s <OffsetValue>     Required for widen|shift|restrict MutaOps.
 -t <TemplateName>    Specify the template to mutate.
 -c <ConfigFile>      Load configuration from config file.
```

Figure A.11. Description of muta.jar

The program *vmodels* translates generated test cases into sequential timed automata (hereafter test automata) by following the approach described in [Hessel et al., 2004], and then embeds the generated automata into each mutated model generated by *muTA*. Each of the generated models is composed by a test automata and a mutated timeliness test model, we use the term *verification model* to refer to these generated models. Figure A.12 shows the parameters accepted by *vmodels*.

```
~$ java -jar vmodel.jar
usage: java -jar vmodel.jar
 -c <ClassPathOfTestCases>   Specify the class path of test cases.
 -i <Mutants>                File/Folder contains mutants.
 -o <PathForVModel>          Specify the path to save vmodels.
```

Figure A.12. Description of vmodels.jar

The program *verifier* finally, determines if each test case may fail or not if executed against a program that behaves as the mutated model. In addition to this, the program *verifier* counts the amount of mutated specifications killed by the test suite. A mutated specification is killed by the test suite when at least one of the test cases belonging to the test suite fails. Figure A.13 shows the usage of *verifier*.

```
~$ java -jar verifier.jar
usage: verifier.jar
 -i <VModels>      VModel file/folder to verify.
 -p <TC-Prefix>    Prefix for the name of test case templates.
 -f                Fast verification(stop when first killed).
 -s                Print verification Statistics.
```

Figure A.13. Description of verifier.jar

A.3 STUIOS Toolset

We have implemented STUIOS as a set of Java program briefly described in the following.

The program *testBuilder* automatically generates a stochastic test suite from a given set of test cases. *testBuilder* implements the algorithm described in Figure 5.8 of Chapter 5. To this end, *testBuilder* interacts with the UPPAAL API in order to automatically identify the potential UIO sequences, and then executes UPPAAL to identify the valid UIO sequences and the probabilities for both the test cases and the PUIO sequences identified.

The program *stochasticOracle* is a wrapper around the R toolsuite [R Core Team, 2013] which is used to calculate the Wilson Score interval and then decide the test execution results. For each executed test case *stochasticOracle* checks if the output corresponds to the output of the stochastic test case, if not, it checks if the output is legal by using the UPPAAL simulator APIs to determine if the timing specifications given by the end users may generate the given output. Finally it determines whether to stop testing or not according to the criterion indicated in Section 5.3.6.

The STUIOS test suite has not been made available to other researchers yet, this is planned in our future work.