Testing Framework for Real-time And Embedded Systems

A Dissertation

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Natural Sciences

to the Department of DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE - INSTITUTE OF COMPUTER SCIENCE

of the Freie Universität Berlin

.

by

-Ing.

Diana Alina Serbanescu

Berlin, April 2013

Supervisor: Prof. Dr.-Ing. Ina Schieferdecker

Second examiner: ______________________

Date of the viva voce/defense: ______________

Declaration of Authorship

I, DIANA ALINA SERBANESCU, declare that this thesis titled, 'REAL-TIME TESTING FRAMEWORK FOR EMBEDDED SYSTEMS' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

"Q: There's no point in arguing that the machine is wholly successful, but it has its qualities. I don't like to use anthropomorphic language in talking about these machines, but there is one quality...
A: What is it?
Q: It's brave.
A: Machines are braver than art."

City Life, Donald Barthleme

FREIE UNIVERSITÄT BERLIN

Abstract

Freie Universität Berlin
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE - INSTITUTE OF COMPUTER SCIENCE

Doctor of Natural Sciences

by -Ing.
Diana Alina Serbanescu

Real-time reactive and embedded systems are usually used in circumstances where safety is important and the margin for error is narrow. These kinds of systems have applicability in a broad band of domains such as: automotive, avionics, air traffic control, nuclear power stations, industrial control, etc. As the name denotes, the main feature of "real-time" systems is the criticality of their timeliness. Guaranteeing a certain timeliness requires appropriate testing. As manual testing is burdensome and error prone, automated testing techniques should be developed. Although the importance of having a standard environment for automatic testing is high, the technologies in this area are not sufficiently developed. This thesis reviews the standardized test description language "Testing and Test Control Notation version 3 (TTCN-3)" as a means for real-time testing and it proposes extensions to enable real-time testing with TTCN-3. The aim is to provide a complete testing solution for automatic functional and real-time testing, built around this already standardized testing language. The solution includes an environment for designing and running the tests written in the extended language. As a proof of concept, test examples, designed using the enhanced TTCN-3, are mapped to real-time platform implementations and the timeliness of each implementation is analyzed.

FREIE UNIVERSITÄT BERLIN

Zusammenfassung

Freie Universität Berlin
FACHBEREICH MATHEMATIK UND INFORMATIK - INSTITUT FÜR INFORMATIK

Doktor der Naturwissenschaften

von -Ing.
Diana Alina Serbanescu

Echtzeit-reaktive und eingebettete Systeme werden gewöhnlich in Bereichen genutzt, in denen Sicherheit sehr wichtig und die Fehlertoleranz begrenzt ist. Diese Systeme finden ihre Anwendung in vielen Bereichen: in der Automobilindustrie, in der Luftfahrtindustrie, in der Luftfahrtberwachung, in nuklearen Anlagen, in der Regelungs- und Steuerungstechnik, etc.

Wie der Name bereits andeutet, liegt das charakteristische Merkmal von Echtzeitsystemen in der Sicherstellung der Rechtzeitigkeit von Ereignissen. Das Garantieren einer bestimmten Rechtzeitigkeit erfordert geeignete Testmethoden. Da manuelles Testen beschwerlich und sehr fehleranfällig ist, sollten automatisierte Testmethoden zur Anwendung kommen. Obwohl es sehr wichtig ist, für das automatisierte Testen eine standardisierte Testumgebung zu besitzen, sind die Technologien in diesem Bereich nicht ausreichend entwickelt. Diese Dissertation betrachtet die standardisierte Testbeschreibungssprache "Testing and Test Control Notation Version 3 (TTCN-3)" als ein Mittel für das Testen von Echtzeitsystemen und unterbreitet Vorschläge für Erweiterungen der Testsprache, um das Testen von Echtzeitsystemen mit TTCN-3 zu ermglichen. Ziel ist es, ein komplette Testlösung für das automatisierte funktionale Testen und das Testen von Echtzeitsystemen bereitzustellen, welche auf diese bereits standardisierte Testsprache aufbaut. Die Lösung bietet Erweiterungen der Sprache sowie eine Umgebung für das Erstellen und das Ausführen der Tests an. Im Rahmen einer Machbarkeitsstudie werden Testbeispiele erstellt, die auf Implementierungen von Echtzeitsystemen angewendet werden.

Acknowledgements

First of all I would like to express my entire gratitude to my Professor Ina Schieferdecker. This thesis would not have been possible without the continuous support, tutoring and encouragements that my Professor, Ina Schieferdecker, offered to me. She provided me not only the opportunity, but also the example in pursuing an academic career: her good technical advices, elegant arguments while discussing technical problems, her professional attitude towards research, attention to detail, but also friendly and relaxed, yet engaging attitude, guided me along the way, and were most valuable for the development of this work.

Secondly, I would like to express my gratitude to Professor Valentin Cristea, who tutored and encouraged me during my university years, and more recently for taking the responsibility of being my second supervisor. I would like to thank him for the prompt support and the valuable and detailed corrections he gave me. These proved very helpful in increasing the quality of my thesis.

I am grateful for having the opportunity of accomplishing this PhD in a very professional and productive environment. The work presented here was realized in collaboration with the MOTION team from Fraunhofer FOKUS, with the support of Technical University of Berlin and with the collaboration of testing team at the Daimler Center for Automotive Information Technology Innovations (DCAITI). Thank you Ina, for giving me the chance to work with and learn from those wonderful people!

I would like to extend thanks to FOKUS Fraunhofer Institute for Open Communication Systems for involving me in interesting research from the automotive domain and for giving me the chance to attend conferences all over the world. During those years I developed a lot, not only as a researcher, but also as a person. Many thanks to the "Alfried Krupp von Bohlen und Halbach-Stiftung" for offering me a two years scholarship, which helped me focus on my study. I express my deepest gratitude to Fraunhofer FOKUS for giving me the chance to start and grow as a researcher. I would also like to address a big "Thank You!" to my wonderful former and current colleagues from the MOTION team, especially to Dr.George Din, Dr.Diana Vega, Andreas Hoffmann, Dr.Alain-Georges Vouffo Feudjio, Marc-Florian Wendland, Dr.Jürgen Gromann ,Dr.Justyna Zander, Dr.Zhen Ru Dai, Petre Razvan, Arun Prakas, Nikolay Vassilev Tcholtchev, Dr.Ranganai Chaparadza and Victoria Molovata for the interesting discussions and new perspectives, for excellent guidance and tips, for making the work fun and making me feel at home within the team. Also I would like to thank to my colleagues from DCAITI for the friendliness and professionalism. Among them I would especially like to acknowledge Ilja Radusch, Horst Rechner, Björn Schünemann, Mihai Rigani and Martin Goralczyk.

As many of the ideas from this thesis were firstly presented and originated in the intense discussions that took place within the context of TEMEA project, I dearly want to thank all the members of the project for offering me a place at the discussion table.

My warmest appreciations go to my dearest friends. I am grateful to Petre Razvan, Bogdan Pintea, Diana Vega, George Din, Madalina and Dirk Tempelmann for being on my side when I needed them, both personally and professionally. Thank you, guys, I wouldn't have made it without you!

I would also like to mention the great support and unconditioned appreciation and encouragement that I got from my dear friend Steve Cosgrove, Lecturer at Whitireia New Zealand. Steve helped me a lot with the proofreading of this thesis. Thank you, Steve for all your help and thank you for never ceasing to believe in me! Despite the distance you were very near all along. Furthermore, I wish to express my enthusiasm and appreciation to all young researchers out there, some of which I had the luck to meet during conferences. It is very encouraging to meet and befriend like-minded people from all over the world.

Last but not least I would like to thank and express my love to my wonderful family. I want to thank to my amazing parents for unconditioned love and understanding. Thank you for handling so well the distance and my decision to come to Germany for study. My dear parents, I am so proud of you and I would like you to be proud of me too. I would also like to thank my uncles and aunts and my grandparents who raised me to be what I am. This work is dedicated to you all!

Last but not least, I would like to thank Neal Cahoon for helping and encouraging me and bearing with me during this period of my life.

Contents

List of Figures

List of Tables

Abbreviations

AEM	**A**dvanced **E**nergy **M**anagement
A/D	**A**nalog-to-**D**igital convertor
ACC	**A**daptive **C**ruise **C**ontrol
ADA	**A**dvanced **D**river **A**ssistance
API	**A**pplication **P**rogramming **I**nterface
ASN.1	**A**bstract **S**yntax **N**otation **.1**
BNF	**B**ackus-**N**aur **F**orm
BSD	**B**lind **S**pot **D**etection
CDS	**C**ollision **D**etection **S**ystem
CP	**C**rash **P**revention
CPU	**C**entral **P**rocessing **U**nit
CS	**C**rash **S**afety
CSP	**C**ommunicating **S**equential **P**rocesses
CTE	**C**lassification-**T**ree **E**ditor
CTM	**C**lassification-**T**ree **M**ethod
D/A	**D**igital-to-**A**nalog convertor
DAS	**D**river **A**ssistance **S**ystem
EBNF	**E**xtended **B**ackus-**N**aur **F**orm
ECU	**E**lectronic **C**ontrol **U**nit
ETSI	**E**uropean **T**elecommunications **S**tandard **I**nstitute
EDF	**E**arliest **D**eadline **F**irst
IDL	**I**nterface **D**escription **L**anguage
IUT	**I**mplementation **U**nder **T**est
HIL	**H**ardware-**i**n-the-**L**oop
MIL	**M**odel-**i**n-the-**L**oop

MMI	**M**an **M**achine **I**nterface
MSC	**M**essage **S**equence **C**harts
MTC	**M**ain **T**est **C**omponent
OEM	**O**rriginal **E**quipment **M**anufacturer
PA	**P**arking **A**ssistance
PAT	**P**rocess **A**nalysis **T**oolkit
PP	**P**edestrian **P**rotection
PTC	**P**arallel **T**est **C**omponent
RMA	**R**ate **M**onolitic **A**lgorithm
RT	**R**eal-**t**ime
RTOS	**R**eal-**t**ime **O**perating **S**ystem
RTSJ	**R**eal-**t**ime **S**pecification for **J**ava
RTSUT	**R**eal-**t**ime **S**ystem under **T**est
RTTS	**R**eal-**t**ime **T**est **S**ystem
SGC	**S**top and **G**o **C**ontrol
SIL	**S**oftware-**i**n-the-**L**oop
SUT	**S**ystem **U**nder **T**est
TA	**T**imed **A**utomata
TCI	**T**TCN-3 **C**ontrol **I**nterface
TIOTS	**T**imed **I**nput-**O**utput **T**ransition **S**ystem
TRI	**T**TCN-3 **R**untime **I**nterface
TS	**T**est **S**ystem
TTCN-3	**T**esting and **T**est **C**ontrol **N**otation version **3**
XML	**E**xtensible **M**arkup **L**anguage
UTC	**C**oordinated **U**niversal **T**ime
V&V	**Verification & Validation**
WCET	**W**orst-**c**ase **E**xecution **T**ime

Dedicated to my wonderful parents.

Chapter 1

Introduction

"If knowledge can create problems, it is not through ignorance that we can solve them."
Isaac Asimov

Real-time systems are those systems in which the correctness of the system depends not only on the logical results of computation but also on the time at which the results are produced [28]. They span a broad spectrum of complexity from very simple micro-controllers in embedded systems to highly sophisticated, complex, and distributed systems. Real-time and embedded systems play an important role in the contemporary world and many industrial processes rely on their good functioning. Their area of applicability is very wide, ranging from industrial process controllers, to technical equipment used in the health, automotive, avionics and space control sectors.

As each sector develops technologically, more and more equipment and functionality is added. For example, in the automotive industry, innovations are realized particulary by electronic systems and software driven functions. In a modern vehicle there may be between 30 and 80 controllers, which communicate with one another over different bus systems [4]. The quality assurance of such systems is very important, since most of the real-time systems perform safety or other critical procedures, which may lead to fatality if not performed correctly.

Testing plays an important role in asserting the quality of an application and should usually accompany the most part of the development process - at different stages - and also early in the life of a product. The challenge that has to be assumed is to be able to manage the increasing complexity of interacting real-time and embedded systems, and to perform an accurate and precise testing of their functionalities, before their release for use.

There are several types of testing that should be performed on a large software system: unit, integration, functional, system, acceptance, beta and regression. Each type of test has a *specification* that defines the correct behavior that the test is examining, so that incorrect behavior, *an observed failure*, can be identified. Based on opacity of the tester's view of the code, these types can be divided into two basic classes of software testing, *black-box* testing and *white-box* testing. *Black-box* testing, also called functional testing, is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. *White-box* testing, also called structural testing and glass box testing, is testing that takes into account the internal mechanism of a system or component.

In the language of V&V, *black-box* testing is often used for *validation* - *"are we building the right software?"* - and *white-box* testing is often used for *verification* - *"are we building the software right?"*. Because the goal is to verify and validate already developed *real-time* applications, which come from different manufacturers, the solution presented here focuses on *black-box* testing [25]. Therefore, whenever *testing* is mentioned in the following, it should be understood as a substitute for *black-box testing*. A more comprehensive description of the different types of testing and of the relevance of *black-box testing* in the context of V&V can be found in [11].

1.1 Scope Of The Thesis

Testing of *real-time and embedded applications* represents a challenge due to the special nature of such systems, requiring not only to test the functional aspects, but also *timing aspects* of the computation. In addition to these, other characteristics of the *real-time and embedded applications* - the most relevant being summarized in Table 1.1 - have to be taken in consideration in order to be properly addressed by the testing process.

timeliness requirements
complexity, diversity, heterogenous
rapid development process
various manufacturers
interoperability requirements
functional in different environments

Table 1.1: Key Characteristics Of The Real-time Systems That Should Be Taken Into Account By Testing

The challenge is even greater when we face the *complexity* and *diversity* already mentioned. Furthermore, an *automatized* testing process will increase the efficiency of testing and will be more suitable for the covering and testing requirements of such *heterogenous* systems. Compared with traditional testing methods, which are usually involving a lot of manual testing, an automatized testing is less prone to errors.

Test cases should be written in an *easy-to-read* manner to minimize duplicate specifications. The tests should be used to provide a base-line for *regression testing*. Hence, repeatability is a key property. The tests should be able to execute in several environments and therefore, for increased portability, they should carry a minimum of environment-specific details [38].

Unfortunately, testing for real-time and embedded systems is still lacking complete automation technique. It is usually performed by a multiplicity of proprietary test systems and test platforms, lacking a *standardized* approached which can unite the different visions. A comprehensive state of the art of these testing solutions is made in Chapter 3. Nevertheless, none of the existing frameworks have been based on a standardized test language that can provide a common platform for test collaboration and test case exchange. Their evolution was heterogenous and the demand for a common and standardized approach was the next requirement that still needed to be addressed.

With respect to these aspects, the aim of this thesis is: *To build a manufacturer-independent testing framework that combines functional test automation with real-time test automation by means of a standardized testing specification language.*

In order to realize the proposed testing framework for real-time, this thesis focuses on the following aspects:

Using the Testing and Test Control Notation, version 3 (TTCN-3). The standardized testing language that remains at the basis of this testing framework is TTCN-3. As many of the test tool providers from the industry (e.g. [23], [24]) are pitching in its favour, there are plenty reasons for choosing TTCN-3 (a more detailed argumentation is given in Chapter 3). But the most significant one is that TTCN-3 is actually *the only standardized test technology enabling test automation.* Since its creation, TTCN-3 has been successfully applied into industrial projects, ranging from areas as telecommunication, automotive domain, to healthcare and much more. Being an international, open and maintained standard with standardized interfaces, extensibility is built in. Permanent updates ensure its usability according to the latest testing requirements.

Besides typical programming constructs, TTCN-3 contains all the important features to specify test procedures and test campaigns for all kinds of testing, such as functional, conformance, interoperability, or load tests. These test-specific features are unique compared to traditional scripting or programming languages, and above all remain technology-independent. TTCN-3 defines test cases on an abstract level and supports full test execution control [24].

Taking the standard and making it appropriate for testing real-time applications seems, in this context, a natural step forward. Unfortunately, TTCN-3 was not developed with real-time focus in mind and it lacks the mechanism for dealing with real-time specific test situation. Therefore, the challenge assumed in the context of this thesis is also to enhance TTCN-3 with language extensions, reflecting all the needed real-time testing concepts that have been missing from the language.

Real-time testing concepts. In addition to the specific functional requirements defined by their specification, real-time systems also have to respect special requirements for timing [111]. In order to ensure the timeliness of the *SUT*, the *TS* ought to be time-predictable as well. Assuring this timeliness for the *TS* starts from the test design. After investigating the requirements specific to real-time applications and the problems that arose due to the particularities of those requirements, a number of concepts specific to real-time test design were identified. Examples of such concepts are shortly summarized in the following.

Each *TS* for real-time should have *a clock* capable of a desired precision, *a function to read the clock*, a set of *data types for representing absolute and relative time values* for saving values read from the clock, mechanisms associating *timestamps* with important events (e.g. incoming of messages) during the test execution. As reading and saving time values of a clock would be not enough for assuring the timeliness of the *TS*, mechanisms for associating durations with test behavior should also be at hand. These should enable the test designer to *delay the execution of certain parts of test behavior* for a specified period of time or *resume blocking instructions* (e.g. awaiting incoming messages on ports) after a maximum time frame.

Real-time testing extensions for TTCN-3. Syntactic and semantic definitions. As previously mentioned, in order to achieve the goal of this thesis, TTCN-3 was extended with a minimal and yet complete set of constructs for real-time testing, reflecting the concepts discussed in the previous paragraph. The research made for the development of these constructs includes an extensive study of real-time programming languages - with focus on their particular features for real-time - and the revision of previous extensions proposed for TTCN-3.

In the context of this thesis, the enhancements to TTCN-3, needed for real-time testing, are the following: data types suitable for expressing time values (*datetime, timespan, float, tick*), special operations relying on time (*now, wait, testcasestart, testcomponentstart, testcomponentstop*), *timestamp* for incoming or outgoing communication, time restrictions for message receival using time predicates (*at, within, before, after*), inducing events (e.g. sending of a message, starting a test component, etc.) at established time points using the time predicate *at*.

The new proposed concepts are integrated into the syntactical structure of the TTCN-3 language, by means of clear syntactic rules, based on extended Backus–Naur Form (BNF) notation. The semantics of the real-time *TS* realized on the basis of enhanced TTCN-3 is further defined by means of timed automata [92]. For each TTCN-3 instruction that relies on a real-time concept, the associated semantic is represented as timed automata. This approach is new and different from the way semantics of TTCN-3 was previously defined into the standard. The motivation for choosing timed automata is that they are mathematical instruments for modeling timed behavior in a formal way. This approach also opens new and interesting possibilities, as, for example, semantical validation of the timed TTCN-3 code, based on model-checking methods developed for timed automata.

Real-time testing framework architecture. The next step was to define the architecture of the real-time testing framework and explain the algorithms for implementing the proposed concepts on real platforms. Based on those, a set of design patterns were defined in order to draw the connection between the newly introduced test concepts and real-time operating systems services. In this way, any real-time operating system presenting the required services, could realize the behavior implied by a certain real-time test concept.

Benchmark of the real-time testing framework. Real-time case study. The TTCN-3 extensions for real-time were implemented on a real-time operating system (Linux with RTAI) for a proof of concept. Delays and latencies were measured for this implementation in benchmark scenarios. Evaluation of the results led to the conclusion that the goal was reached. The test framework for real-time realized has worse case execution times (WCETs) bounded in the range of microseconds, with the condition that the testing behavior is consisting of a real-time schedule-able set of tasks (schedulability tests for a set of tasks are discussed in [22]).

In the second case study, the real-time testing concepts were used in combination for building a *TS* for an application from the automotive domain. The application consists of an Electronic Control Unit (ECU) controlling the functionality of a car door. Differently triggered sequences of functionality are required to be performed with respect to strict time constraints. The real-time test framework was efficient in asserting conformance to both functional and timing requirements, with a required precision, for the timing requirements, in the range of microseconds. Some failures in the behavior of the *SUT* were discovered for specific sets of inputs.

The work presented here stems from the author's participation in the TEMEA [79] project, on the basis of which, the additional standardized extension for TTCN-3 [80], regarding real-time and performance, was published. Nevertheless, the concepts introduced in this thesis are a wider, more comprehensive and more complete set compared to the concepts developed by TEMEA, or the concepts adopted into the standard.

Before a deeper insight is found in the proposed design for a testing framework for real-time and embedded systems, one should examine the following questions carefully:

Q1: *What does black-box testing for real-time mean in comparison to traditional black-box testing of systems?* Black-box testing is a method of software testing that tests the functionality of an application based on inputs and outputs provided or acknowledged to or from the respective application. Specific knowledge of the application's code in general is not required. Test cases are built around *specifications and requirements* which are describing what the application is supposed to do. Therefore, the above question can be translated into the following one:

Q2: *Which specifications and requirements are particular to real-time and embedded systems and how are those requirements influencing the testing process?* A system is said to be real-time if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed. This means that beside the usual functional requirements, there are also some *timeliness requirements* that should be respected by the analyzed system. Special test cases should be designed for verifying whether those *timeliness requirements* are respected or not.

Q3: *What testing frameworks for real-time are out there? What is the advantage of the presented approach over the existing solutions?* A state of the art ought to be made for presenting the existing tools for testing real-time and embedded systems. Different features of those languages ought to be analyzed and their advantages and disadvantages ought to be presented. The need for a new approach should result from the insufficiency of the existing tools to cover important aspects of testing real-time systems.

Q4: *What test specification languages are out there and which one is most suitable as a basis for a testing framework for real-time? Is there any standardized testing specification language that can be used?* A state of the art ought to be made for presenting the existing test specification languages. Different features of these languages ought to be analyzed and then, based on several criteria - e.g. usability, standardization, etc. - a selection ought to be made. TTCN-3 was found to be the best match for the envisioned goal. The main reason to select this language lies in the fact that TTCN-3 is at present the only standardised test language and technology-enabling test automation. Since its creation, it has become popular in the industry and has been successfully used for various application domains like telecommunications, automotive, health care and more. Additionally, various features offered by this language make it a suitable technology to build upon a testing framework for real-time. All these are going to be presented in more detail in Chapter 3.

Q5: *How suitable is TTCN-3 for designing test cases for verifying timeliness requirements*? Although TTCN-3 is a well structured and modular language, specially conceived for black-box testing industrial application, with powerful tools, such as sending and receive operations on ports and matching mechanism against predefined templates, it was not thought-out with real-time target systems in mind.

Q6: *What does TTCN-3 lack for being appropriate for real-time testing? How can the shortcomings be overcome?* In order to move the accent from a purely functional form of developing tests towards an approach where time aspects are equally important, the testing language should possess good mechanisms and concepts for dealing with time.

Q7: *What makes a programming language real-time? What characteristics and concepts are specific to a real-time programming language? Are those characteristics applicable for a testing language as well?* Several real-time languages need to be studied in order to draw some answers to those questions and to find a set of concepts that are necessary both for designing and testing real-time applications and that have no equivalent in the current TTCN-3 standard.

Q8: *How can the new concepts be integrated into TTCN-3 from a syntactical and semantical perspective?* For each proposed concept, syntactical and semantical definitions ought to be designed for integrating them into the language and establishing the relationships with the other elements of TTCN-3.

Q9: *How can these new concepts be implemented?* TTCN-3 is a specification language. This means that test specification should be translated in a code that can be executed. Therefore, design patterns for implementing the new concepts need to be introduced.

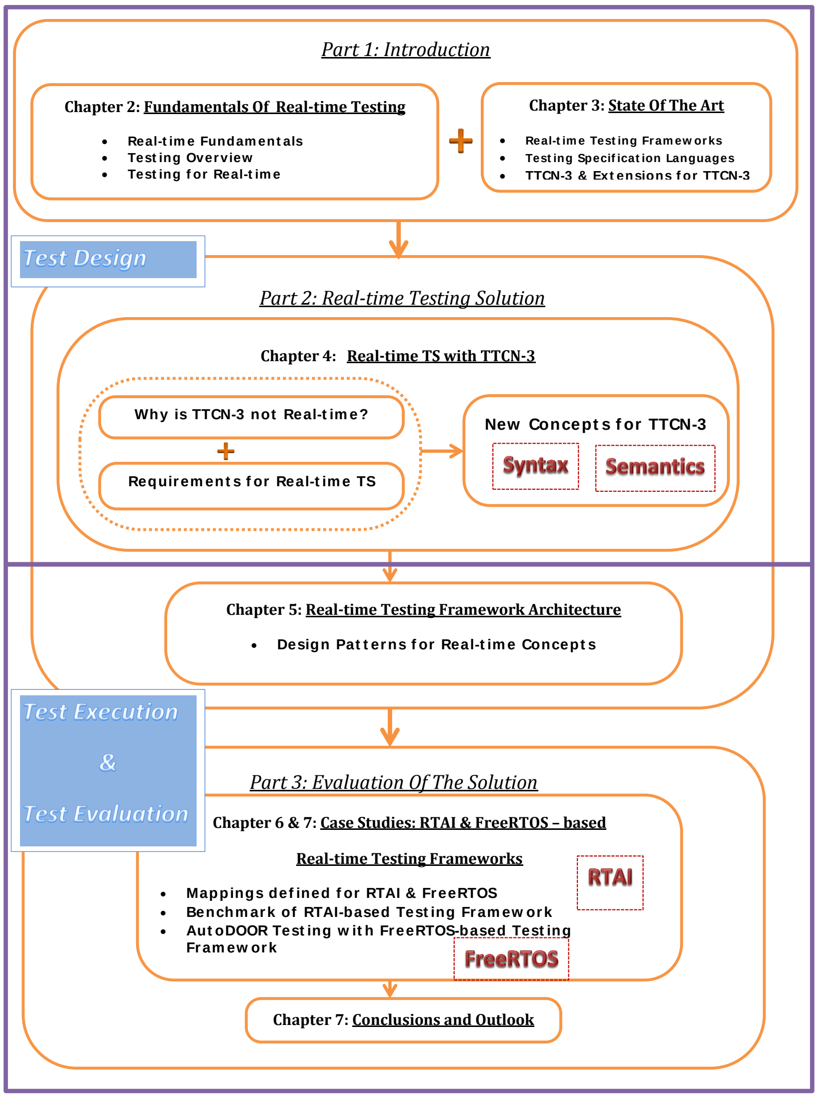
Q10: *What real-time platform(s) is targeted for the proof of concept? What are the main criteria for selecting the real-time operating system?* There are a wide variety of real-time operating systems. In order to implement some abstract test concepts, a real-time operating system should be chosen. The real-time operating system should be selected on the basis of a list of criteria.

Q11: *How would a mapping between the new introduced concepts and the API, provided by the selected real-time operating system, look like?* A future aim would be that the real-time test cases are to be automatically translated into code that can be executed on the selected platform. Therefore, it is necessary that mapping rules are established for the automatic translation of test cases.

Q12: *How should the* TS*, provided by this solution, be evaluated as suitable for testing real-time applications?* After the implementation of the new concepts, a benchmark of the resulting *TS* should be performed. The benchmark will be based on worst case execution times (WCETs) for a variety of test cases, defined using the newly introduced test concepts. The aim is to cover and evaluate most of the possible usage situations for the new concepts within TTCN-3.

All the afore mentioned questions are going be answered, one by one, during the forthcoming chapters of this thesis. The roadmap and the dependencies among the chapters are presented in the following Sections.

1.2 Structure Of The Thesis


Part 1: Introduction
Chapter 2: Fundamentals Of Real-time Testing
Real-time Fundamentals
Testing Overview
Testing for Real-time
Chapter 3: State Of The Art
Real-time Testing Frameworks
Testing Specification Languages
TTCN-3 & Extensions for TTCN-3
Test Design
Part 2: Real-time Testing Solution
Chapter 4: Real-time TS with TTCN-3
Why is TTCN-3 not Real-time?
Requirements for Real-time TS
New Concepts for TTCN-3
Syntax
Semantics
Chapter 5: Real-time Testing Framework Architecture
Design Patterns for Real-time Concepts
Test Execution & Test Evaluation
Part 3: Evaluation Of The Solution
Chapter 6 & 7: Case Studies: RTAI & FreeRTOS – based Real-time Testing Frameworks
Mappings defined for RTAI & FreeRTOS
Benchmark of RTAI-based Testing Framework
AutoDOOR Testing with FreeRTOS-based Testing Framework
RTAI
FreeRTOS
Chapter 7: Conclusions and Outlook

.

FIGURE 1.1: Dependencies Between Chapters

In Figure 1.1 can be visualized the structure of this thesis, together with the connections between chapters. There will be a logical partitioning of this material into three main parts:

Part 1 – containing **Chapters 1-3** – presents the fundamental knowledge that is necessary for understanding the proposed solution, of testing real-time systems in a standardized testing environment.

Chapter 1: presents the scope and structure of this work together with a short presentation of each of the following chapters.

Chapter 2: holds theoretical mechanisms necessary for understanding the rest of the thesis. The main topics in discussion are: fundamentals of software V&V and presentation of black-box testing in this context, testing concepts for real-time with regard to the characteristics and specific requirements of real-time systems, real-time programming language concepts and specific features, real-time operating system concepts, semantic models for real-time and testing with respect to real-time applications. The information contained in this chapter provides answers to the questions **Q1,Q2 and Q7**, formulated in the previous section, 1.1.

Chapter 3 : begins with presenting the state of the art for real-time testing frameworks, listing already existing theories and solutions. Advantages and disadvantages of these frameworks are thoroughly discussed. The flaws of the existing solutions, with regards to the current need for testing in the real-time systems world (e.g. lack of automation, proprietary technology lacking a standardized approach), are emphasized here. The solution adopted by this thesis, presented in this context, as an answer to these needs, overcomes the weaknesses of the other approaches. In the second part of this chapter, a state of the art of the existing test specification languages is realized. Those specifications are compared, based on a list of criteria (e.g. modularity, portability), and the most suitable one is chosen as a basis for the test framework presented here. TTCN-3 is selected as the appropriate choice. Suitability of TTCN-3 with regard to testing real-time system is discussed. Some previous endeavors using TTCN-3 specification language for designing tests in the real-time context are considered further. Positives and negatives of the preceding approaches are shortly analyzed and discussed. This chapter adds more information and provides answers to the questions **Q3 to Q6** from Section 1.1.

Part 2 – containing **Chapters 4-6** – encompasses the solution of this thesis and provides the answers to the questions **Q6 to Q10**. In order to properly understand the solution presented here, the reader needs to be familiar with the theoretical aspects presented in **Part 1**.

Chapter 4: starts with presenting the problem in a formal manner. The deficits of TTCN-3 regarding real-time are summarized. A formal definition of a *TS* for real-time is given here by means of an extended timed automata (TA) model. New concepts are added to TTCN-3 specification language for solving the presented shortages of the language (lack of appropriate data types for storing time values, inaccurate timers based on snapshot semantics, lack of means for controlling incoming and outgoing communication or for inducing events at specific times, lack of means to constrain test behaviour). These concepts are integrated syntactically into the language by describing each particular syntactic rule that should be added to the grammar of TTCN-3. The semantics of each concept is presented using the extended TA model that was introduced in the beginning. For a good understanding of this chapter, one has to be familiar with the notions introduced in Chapters 2 and 3.

Chapter 5: will show some design patterns for implementing the new concepts introduced in **Chapter 4**.

Part 3 – contains **Chapters 6, 7 and 8** – presents two case studies with two concrete realizations of the concepts based on two real-time operating system platforms. It also draws together the evaluation of the presented solution, explaining the conclusion and proposed prospects for the future.

Chapter 6: relies on the knowledge cumulated from **Chapter 4** and **5**. Here is shown how the new concepts for real-time introduced in **Chapter 4** are mapped to real-time platforms. A real-time operating system selection is realized on the basis of several criteria (e.g. design, supported languages, etc.). A wide range of real-time operating systems are investigated and two of them are chosen as proof of concept: Linux with RTAI and FreeRTOS.

Chapter 7: In benchmark scenarios, worst case execution cases are calculated for each new concept introduced in **Chapter 4**. The benchmarking was realized for the RTAI implementation. The FreeRTOS implementation was used in a case study, presenting an *SUT* consisting of a simple automotive application: a controller for an automatic car door. The FreeRTOS-based *TS* was successful in testing the *SUT* and some failures of the *SUT* were identified.

Chapter 8: presents the conclusions and outlook for further work. This thesis represents only a step forward, not the end of the journey.

Appendixes – Containing: **Appendix A, Appendix B, Appendix C, Appendix D, Appendix E** – comprehends completion of the theoretical and solution chapters.

Appendix A contains the syntax for the real-time extensions of TTCN-3.

Appendix B contains the functions designed to realize the conversions between the different formats proposed for representing time.

Appendix C contains the semantics for time expressions with numerical and logical operators.

Appendix D contains the complementary logic rules to the semantics of the new real-time extensions for TTCN-3.

Appendix E contains the real-time extensions for TTCN-3, introduced in Chapter 4, by means of examples.

1.3 Dependencies Of Chapters

Chapters 4, 5 and 6 use concepts presented in Chapters 2 and are relying on the states of the art presented in 3. Nevertheless, a reader who is familiar with the concepts related to real-time systems and real-time testing may proceed directly to Chapter 4.

Chapter 4 describes the adopted real-time extensions for TTCN-3, on the basis of which the real-time testing architecture from Chapter 5 is built upon. These concepts and this architecture are used for implementing the case studies presented in Chapters 6 and 7. A reader who desires a quick overview on the topics of this thesis should read the summaries provided at the end of each chapter, as well as the overall summary and conclusion given in Chapter 8.

Chapter 2

Fundamentals Of Real-time Testing

"I know nothing except the fact of my ignorance."
Socrates

The testing of real-time systems is a complex conceptual and technical activity that starts with a good understanding of the real-time systems and their specific requirements. The purpose of this chapter is to provide the theoretical foundation required for a good understanding of the testing solution adopted in this thesis. Therefore, the most important concepts of real-time that are also relevant for testing are displayed here.

This chapter is split into two sections. The first section contains an overview of the real-time systems, including a taxonomy of real-time and a discussion about specific requirements for real-time. Knowing to identify different categories of real-time systems and understanding their particular requirements is important for designing suitable tests. Adjacent aspects as programming languages for real-time, real-time operating systems (RTOSes), semantic models and tools for describing and verifying real-time applications are further addressed here. Concepts from real-time programming languages are reused to define real-time test specifications (e.g. *clocks*, *delays*, *timeouts*, *deadlines*, etc.), semantic models for real-time are used to define and validate real-time test specification, and RTOSes are part of the platforms on which the tests are run. The second part focuses on black-box testing in the context of real-time and defines real-time test specific concepts as *what is a real-time test system (RTTS)* or *what is a real-time system under test (RTSUT)*.

The concepts presented in this chapter are being used in Chapters 4, 5 and 7.

2.1 Real-time Fundamentals

Real-time systems are those systems in which *the correctness of their execution depends not only on the logical results of computation but also on the time at which the results are produced* [40]. They span a broad spectrum of complexity from very simple micro controllers in embedded systems – e.g. a microprocessor controlling an automobile engine – to highly sophisticated, complex, and distributed systems – e.g. air traffic control.

It can be argued that any practical system can be regarded as real-time. Even a web-browser should be able to respond to commands within a reasonable amount of time (e.g.

1 second), or it would become torturous to use [41]. But there is still a big difference between real-time influencing usability and user-friendly features of an application, and real-time deadlines that are critical and very important with respect to a high and sometimes fine-grained precision.

Thus, the range of real-time application can be classified into four main categories, which determine how critical the real-time constraint is:

1. *Soft real-time* systems are systems in which the performance is degraded but not destroyed by failure to meet response time constraints (e.g the web-browser given as a previous example will belong to this group).
2. *Hard real-time* systems on the other hand are systems in which failure to meet a single deadline may lead to complete and catastrophic system failure (e.g. a power plant controller or an electronic control unit (ECU) controlling the functionality of the engine in a car, etc.).
3. *Firm real-time* systems are placed somewhere in between the two above mentioned categories. A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete and catastrophic system failure (e.g. a navigator system inside a car can be such a firm real-time system, as the driver can, for example, miss the exit from the highway if the route is not calculated in time).
4. *Real real-time* systems are systems which are hard real-time and for which the response times are very short (e.g. Missile Guidance System, etc.).

2.1.1 Real-time Systems Requirements

Real-time systems have *end-to-end time requirements* that add to the general constraints for the execution. That means that precise portions of execution have to be accomplished in predefined time frames.

Real-time systems need to have *fast, predictable, and bounded responses* to all types of *events* such as interrupts and messages, and also the ability to manage system resources to meet processing *deadlines.* A *deadline* represents a time within which a task or a computation should be completed.

Two ways in which this works concerns *event-driven* systems and *time-driven* systems:

1. *Event-driven* systems will respond to external stimuli that could occur at any point in time. This is usually the case because the system is sensing something in the

real world that is inherently unpredictable. The *performance requirement* for this type of system says: *If event x happens, you must respond appropriately within n milliseconds.*

2. *Time-driven* systems, on the other hand – *their actions driven by the passage of time or the arrival of absolute points in time* – have certain tasks that must be accomplished on a periodic basis that is typically known ahead of time. The *performance requirement* of this type of system sounds like: *Every n milliseconds, you must perform such and such operation.*

In practice, nevertheless, real-time systems often have both event-driven tasks and periodic/time-driven tasks [15].

Timeliness - Deadlines should be met or an appropriate action should be taken. The environment requires a response from the system within a fixed amount of time.
Schedulability - To schedule the computations on the available resources in such a way that deadlines are guaranteed to be met.
Reliability - In many real-time applications, it is important that the system is robust and fault-tolerant.

Table 2.1: Important Characteristics Of Real-time Systems

On a system with shared resources, the responsibility of managing the running tasks on the resources so that the deadlines are guaranteed belongs to the *scheduler*. A system is said to be *schedulable* if it can guarantee that it will meet its performance requirements. The timing requirements of the system must be preserved even at high degrees of resource usage [21].

A real-time system should be *robust* in handling the unpredictable events and no matter what happens, it should always keep one of its properties in predictable terms: *timeliness*. When the system is overloaded by events and it is impossible to meet all the deadlines, the deadlines of critical tasks must still be guaranteed [21].

A property can be *predictable* to the degree that it is known in advance. Hence, at one endpoint of the predictability scale is *determinism*, in the sense that the property is known exactly in advance. At the other endpoint is *maximum entropy*, in the sense that nothing at all is known in advance about the property.

Predictability of timeliness is the most fundamental property of a real-time system [42]. The basic features that a real-time system must demonstrate are summarized in Table 2.1:

2.1.2 Real-time Programming Languages

In [43] it is discussed how important characteristics (see Table 2.1) of real-time have influenced the design of real-time programming languages and real-time operating systems. Ada, Real-time Euclid, Real-time Java, Flex and Real-time C/POSIX are the real-time programming languages whose features adapted for real-time programming are chosen as a basis for discussion in this section.

To ensure that a program meets its specifications, a real-time programming language must allow:

- programmers to express different types of *timing constraints*
- compilers to check the *feasibility of meeting the timing requirements*
- systems to *enforce timing constraints* either before or at runtime

In order to control these timing aspects of the computation, the main mechanisms needed by a real-time programming language are:

- access to a *clock*
- the ability to *delay* execution of a task
- the ability to recognise *timeouts*

As stated in the previous enumeration, in addition to *clock access*, processes must also be able to *delay* their execution either for a relative period of time or until some time in the future. To eliminate the need for busy-waits, most languages and operating systems provide some form of *delay primitive*. Both *relative* and *absolute delay mechanisms* are needed. Absolute delays allow cumulated *drifts* to be avoided.

Ada is a modern programming language that puts unique emphasis on, and provides strong support for, good software engineering practices that scale well to very large software systems. Among its most relevant language features, counts: modularity, a strong, static and safe type system, readability and portability.

Ada also has powerful specialised features supporting low-level programming for realtime, safety-critical and embedded systems. Such features include, among others, *machine code insertions*, *address arithmetic*, *low-level access to memory*, *control over bitwise representation of data*, *bit manipulations*, and a well-defined, statically provable *concurrent computing model called the Ravenscar Profile.* The Ravenscar profile is a subset of the Ada tasking features, designed for safety-critical, hard real-time computing [20] and defining the invocation, synchronization, and timing of parallel tasks [44].

Several vendors provide Ada compilers accompanied by minimal run-time kernels suitable for use in certified, life-critical applications. It should come as no surprise that Ada is heavily used in the aerospace, defense, medical, railroad, and nuclear industries (e.g. "on one new large airplane, approximately one-third of the applications are in Ada" [18].). Ada 95, including its Real-Time System Annex D, has probably been the most successful real-time language, in terms of both adoption and real-time technology.

Real-Time Euclid is a language designed specifically to address *reliability* and *schedulability* issues in environments with *real-time constraints.* The language definition forces every construct in the language to be *time-* and *space-bounded.* These restrictions make it easier to estimate the execution time of the program, and facilitate scheduling to meet all deadlines. Therefore, Real-Time Euclid programs can always be analyzed for schedulability [45].

Real-Time Euclid is modular, procedural, and strongly typed. It is structured and small enough to be remembered in its entirety, thus facilitating programming-in-the-small. Modularity and separate compilation make Real-Time Euclid a suitable language for programming-in-the-large [19]. In contrast to Ada, which has been widely applied in the industry, Real-Time Euclid is an experimental programming language, invented as a tool to be used to predict whether or not real-time software will adhere to its critical timing constraints. It has been demonstrated that this task may be made considerably easier with Real-Time Euclid [17].

Real-Time Specification for Java (RTSJ) specifies how Java systems should behave in a real-time context and has been developed over several years by experts from both the Java and real-time domains. The RTSJ introduces several new features to support real-time operations. These features include new thread types with new scheduling techniques, new memory-management models, new *time*-related classes (e.g. high resolution timers) and operations, asynchronous events and even handlers and other newly introduced frameworks [46].

Java platform's promise of *Write Once, Run Anywhere*, together with the Java language's appeal as a programming language, offer a great cost-saving potential in the real-time. Real-time Java has been successfully deployed in various commercial and defense applications: e.g. the mission planning software for the Boeing J-UCAS X-45C unmanned aircraft was written in Java language and deployed on a real-time virtual machine. French Military FELIN Project by Sagem also uses real-time execution of Java, Aegis Warship Software Upgrade [16], etc.. In successful projects, the choice to use Java has reduced certain risks and demonstrated concrete benefits. In comparison to C, Java offers improved developer productivity, more flexible and general architectures, better portability and scalability, and lower cost maintenance.

Real-time C/POSIX defines a standard way for an application to interface with the operating system. The original POSIX standard has defined interfaces to core functions such as file operations, process management, signals, and devices.

Of the more than thirty POSIX standards, there are seven standards relevant to the development of real-time and embedded systems. The first three standards-1003.1a, 1003.1b, and 1003.1c-are the most widely supported. POSIX 1003.1a defines the interface to basic operating system functions, and was the first to be adopted in 1990.1. Real-time extensions are defined in the standards 1003.1b, 1003.1d, 1003.1j, and 1003.21. However, the original real-time extensions, defined by 1003.1b, are the only ones commonly implemented [47].

The following features constitute the bulk of the features defined in POSIX 1003.1b: *timers* (periodic timers, delivery is accomplished using POSIX signals), *priority scheduling* (fixed priority preemptive scheduling with a minimum of 32 priority levels, *real-time signals* (additional signals with multiple levels of priority), *semaphores*, *memory queues*, *shared memory*, *memory locking* [14].

Commercial support for POSIX varies widely. Because POSIX 1003.1a is based on UNIX, any UNIX-based operating system will be very close to the standard. Some example of operating systems providing 1003.1b are Lynx/OS (used for developing soft real-time systems), VxWorks (with a large spectrum of use, e.g. Mars Pathfinder Mission [13], etc.), Solaris, Linux, QNX, etc..

Tables 2.2 and 2.3 provide an overview of the mechanisms that each of the above summarized programming languages are exhibiting, with regards to basic real-time requirements. Those mechanisms presented here are going to be the focus of enhancement for real-time of TTCN-3 specification.

The three concepts of *clocks*, *delays* and *timeouts* are exemplified in each language. From Table 2.2 it is notable that each of the analyzed languages, with the exception of Real-Time Euclid (who was developed for hard real-time only) are providing a general purpose *clock* library and a *real-time clock* library of high-resolution.

Table 2.2: Characteristics Of The Real-time Programming Languages. Clocks.

Language Feature	**Clocks**	
Programming Language	Language Construct	Description
Ada	`Calendar time`	- wall-based clock, has no requirement to be synchronized with UTC.
	`Real-time`	- based on a monotonically non-decreasing clock whose epoch is system start-up.
Real-Time Euclid	`realTimeUnit`	- this global variable must be initialized, e.g. `realTimeUnit := 1.0 % time unit = 1 seconds`.
	`Time`	- built-in function that returns the time elapsed from the startup of the system to the present in real time units.
Real-Time Java	`java.lang.System. currentTimeMillis`	- wall clock, defined to count millisecond since a defined epoch (1/1/1970 UTC).
	`HighResolutionTime`, `AbsoluteTime`, `RelativeTime`, `Clock`	- those classes add real-time clocks with high resolution time types. The real-time clock is monotonic, and counts milliseconds and nanoseconds since 1/1/1970 UTC.
Real-time C/POSIX	`calendar`, `time_t`	- ANSI C has a standard library for interfacing to `calendar` time. This defines a basic time type `time_t` and several routines for manipulating objects of type time.
	`Real-time POSIX timespec`	- High resolution clock (requires at least one clock of minimum resolution 50 Hz (20ms)), counting the seconds since 1970.

Table 2.3: Characteristics Of The Real-time Programming Languages. Delays And Timeouts.

Language Feature	Language Construct	
Programming Language	**Delays**	**Timeouts on Actions**

Ada	`delay`
	`delay until`
		`select`
		`    delay 0.1;`
		`then abort`
		`   -- action`
		`end select;`

Real-Time Euclid	`atTime`	`atTime`

Real-Time Java	`sleep`	`Timed extends`
		`AsynchronouslyInterruptedException`

Real-time C/POSIX	`timer & signals`
	`nanosleep`
		`timer & signals`

As with `delay`, the construct of `delay until` in Ada, is accurate only in its lower bound. Real-time Java's `sleep` can be relative or absolute. POSIX requires use of an absolute `timer` and `signals`. The C/POSIX `nanosleep` is similar to the `delay` in Ada, with the difference that it may return early due to signal.

Timeouts can be added to any condition, synchronisation primitive, or on message passing. It is also possible, within Ada, to program timeouts or execute timed entry call on protected objects or on passing of messages, by using `delay` statements together with `select`. The exemplified *timeouts on actions* mean that if the action takes too long, the triggering event will occur and the action will be aborted. This is an effective way of catching run-away code. Real-time Java also has overrun handlers. With Real-Time Java, timeouts on actions are provided by a subclass of `AsynchronouslyInterruptedException` called `Timed`. It is also possible, within Ada, to do timed entry calls as shown in the following example:

```
select
    acceptCall(T : temperature) do
        New_temp:=T;
    endCall;
or
    delay10.0;
        --action for timeout
end select
```

While Ada, Real-Time Euclid and Real-Time Java are examples of higher level real-time programming languages, C/POSIX is a lower level language, and its mechanisms are closer to the services of the underlying real-time operating system. In the following Section 2.1.3, the concepts for real-time operating systems are discussed, together with the main scheduling algorithms for real-time.

2.1.3 Real-time Operating Systems

Development of real-time software differs from the development of non-real-time software because execution time must be considered. Predictability is extremely important in real-time programming, and to get it, one needs to keep track of time. The following are the issues that real-time programmers handle:

- *How long will each task take to complete?*
- *How soon can another task be scheduled and start to run?*
- *Which task is most important?*
- *What happens if one task takes too long?*
- *Do the tasks have to communicate, and if so, how?*

In order to keep track of time, means and mechanisms for interrogating the internal clock and for generating time interrupts, are both necessary [48]. Many real-time systems must do simultaneous tasks, and making these tasks coordinate available resources is one aspect of real-time programming. Available resources might be physical such as hard storage, a printer, an input/output (I/O) port, or they might be logical such as non shareable code segment.

This brings the operating system and the underlying hardware processor architecture (e.g. memory, processor speed, bandwidth, etc.) into play. Operating systems are complex programs that interface hardware with user programs [48]. The advantages of the real-time operating system (RTOS) approach are that it provides operations with uniform access to the underlying hardware and can control the scheduling of multiple tasks which includes their access to the shared resources [91].

To be considered *real-time*, an operating system must have a known maximum time for each of the operations that it performs, or at least be able to guarantee that maximum most of the time. Operating systems that can absolutely guarantee a maximum time for these operations are referred to as *hard real-time*, while operating systems that can only guarantee a maximum most of the time are referred to as *soft real-time*.

2.1.3.1 Real-time Operating Systems Concepts

A RTOS is valued moreso for how quickly and/or predictably it can respond to a particular *event* rather than for the given amount of work it can perform over time.

An important concept in real-time operating systems is the notion of an *event*.

Event represents any occurrence that results in a change in the sequential flow of program execution.

Events can be divided into two categories: *synchronous* and *asynchronous*.

Synchronous events are those which occur at predictable times such as execution of a conditional branch instruction or hardware trap.

Asynchronous events occur at unpredictable points in the flow-of-control and are usually caused by external sources such as a clock signal. Both types of events can be signaled to the CPU by hardware signals.

A key property of any real-time system is its *response time*, i.e. the time it takes for the system to react to some external event under worst case conditions. Two important terms, used to describe the response time, shall be defined here:

Interrupt Latency represents the time it takes from a device asserting an interrupt line until the system dispatching the corresponding interrupt handler (ISR). Interrupt latency is due to both hardware and software factors. Interrupts may occur periodically (at fixed rates), aperiodically, or both.

Context Switch Delay defines the time it takes to schedule a task. It involves the scheduler determining which task to run, saving the current task context and restoring the new one [50].

Jitter is the amount of error in the timing of a task over subsequent iterations of a program or loop.

Real-time operating systems are optimized to provide a low amount of jitter when programmed correctly. That means that a task will take very close to the same amount of time to execute each time it is run.

2.1.3.2 Basic RTOS Kernel Services

The kernel of a RTOS provides the *abstraction layer* that hides the hardware details of the processor from application software – or a set of processors – and upon which the application software will run. In providing this *abstraction layer* the RTOS kernel supplies five main categories of basic services to application layer. These services are described below:

Task Management: These are the most basic category of kernel services, and they contain the ability to launch tasks and assign *priorities* to them. The main RTOS service in this category is the *task scheduler. The task scheduler* controls the execution of application software tasks, based on certain scheduling algorithms (presented in Section 2.1.3.3), and can make them run in a very *timely* and *responsive* fashion.

***Inter-task Communication and Synchronization*:** These services make it possible for tasks to pass information from one another, without danger of that information ever being damaged. They also make it possible for tasks to coordinate, so that they can productively cooperate with one another. Without the help of these RTOS services, tasks might well communicate corrupted information or otherwise interfere with each other.

***Timers*:** Since many embedded systems have stringent *timing requirements*, most real-time operating system kernels also provide some basic timer services, able to execute task *delays* and *time-outs.*

***Dynamic Memory Allocation*:** This category of services allows tasks to share chunks of RAM memory for temporary use in application software, as a means of quickly communicating large amounts of data between tasks.

***Device I/O Supervisor*:** These services, if available, provide a uniform framework for organizing and accessing the many hardware device drivers that are typical of a real-time system [51].

2.1.3.3 Scheduling Algorithms For Real-time

Multitasking is a technique to allocate CPU processing time among several tasks. Systems are classified as *preemptive* or *non-preemptive* depending on whether they can preempt an existing task or not. In a *preemptive system*, each task is given a time slice. *Preemptive* means that a task can be stopped, or another task can be preempted. In a *non-preemptive system* a task must run to completion or until it suspends itself.

The *scheduler*, sometimes called the *dispatcher*, is the part of the operating system that decides who gets to do what and when. It is used to select a process from among those ready to run, schedule time for it on the CPU, and maintain a list of ready processes. The *dispatcher* dispatches jobs to the CPU, using the list created by the scheduler. Most real-time operating systems use a *priority-based preemptive scheduler* to keep the system in order. *Priority-based* means that some type of priority scheme will be used to determine how the schedule is made.

In a *static priority system*, the priorities do not change during run-time. Changing the priority of a task during run-time is supported by some systems, and the algorithms for assigning *dynamic priorities* are different from the ones used for *static priorities*.

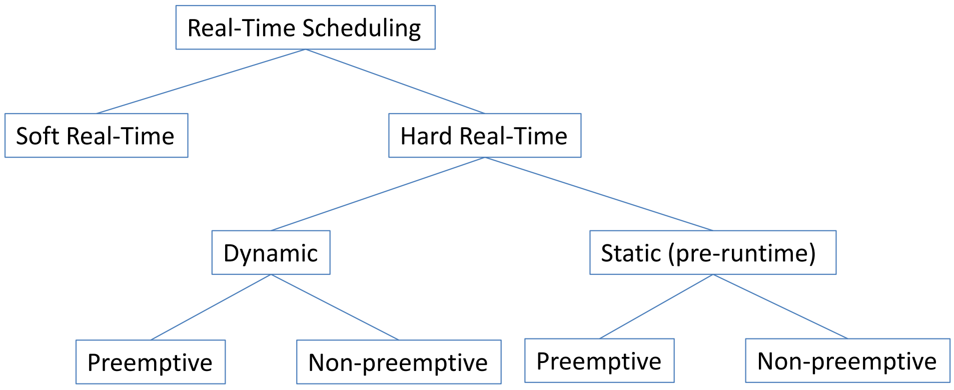


FIGURE 2.1: Taxonomy Of Real-time Scheduling [1]

As shown in Figure 2.1, real-time scheduling can also be categorized into *hard* vs *soft*. Hard real-time scheduling can be broadly classifies into two types, *static* and *dynamic*.

Static scheduling means that the scheduling decisions are made at compile time. A run-time schedule is generated off-line, and is based on the prior knowledge of task-set parameters (e.g., maximum execution times, precedence constraints, mutual exclusion constraints, and deadlines). Therefore, the runtime *overhead* is small. More details on this topic can be found at [52]

Dynamic scheduling implies that the scheduling decisions are made at runtime, by selecting one out of the current set of ready tasks. Dynamic schedulers are flexible and adaptive, but they can incur significant *overheads* because of run-time processing.

Preemptive or *non-preemptive* scheduling of tasks is possible with *static* and *dynamic* scheduling.

The Problem Of Schedulability. In order to certify that a given set of tasks are *schedulable* according to one *scheduling algorithm*, different criteria are defined. The following table present some representative scheduling algorithms, together with their criteria.

Dynamic Algorithms: In Tables 2.4, 2.5 and 2.6 is given an overview of three representative dynamic algorithms, together with their main characteristics and their schedulability criteria for a set of tasks.

TABLE 2.4: Dynamic Scheduling Algorithms. Rate Monotonic Algorithm(RMA).

Algorithm	***Rate Monotonic Algorithm(RMA)***
Scheduling Type	Dynamic Preemptive
Priorities Type	Static Priorities
Description	The **RMA** assigns static priorities based on *task periods*. Here *task period* is the time after which the tasks repeat. The inverse period is *task arrival rate*. For example, a task with a period of `10 ms` repeats itself after every `10 ms`. The task with the shortest period gets the highest priority, and the task with the longest period gets the lowest static priority. At run-time, the *dispatcher* selects the task with the highest priority for execution.
Schedulability Criteria	*According to **RMA** a set of periodic, independent tasks can be scheduled to meet their deadlines, if the sum of their utilization factors of the n tasks is given as follows:* $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$, *where C_i is the computation time, T_i is the release period, and n is the number of processes to be scheduled. For example* $U \leq 0.8284$ *for* $n = 2$ *[53].*

Static Algorithms: In *static scheduling*, scheduling decisions are made during compile time. This assumes parameters of all the tasks are prior known and a schedule based

TABLE 2.5: Dynamic Scheduling Algorithms. Earliest Deadline-First(EDF).

Algorithm	***Earliest Deadline-First(EDF)***
Scheduling Type	Optimal Dynamic Preemptive
Priorities Type	Dynamic Priorities
Description	After any significant event, the task with the earliest deadline is assigned the highest dynamic priority. A significant event in a system can be blocking of a task, invocation of a task, completion of a task, etc..
Schedulability Criteria	*The schedulability test for **EDF** is:* $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$, *where the C_i are the worst case computation times of the n processes and the T_i are their respective inter-arrival periods. **EDF** can guarantee that all deadlines are met provided that the total CPU utilization is not more than 100%. So, compared to fixed priority scheduling techniques like **RMA**, **EDF** can guarantee all the deadlines in the system at higher loading.*

TABLE 2.6: Dynamic Scheduling Algorithms. Priority Ceiling Protocol (PCP).

Algorithm	***Priority Ceiling Protocol (PCP)***
Scheduling Type	Dynamic Preemptive
Priorities Type	Dynamic Priorities
Description	**PCP** is used to schedule a set of dependant periodic tasks that share resources, protected by *semaphores*. The shared resources, e.g. common data structures, are used for *interprocess communication*. The sharing of resources can lead to *unbounded priority inversion*. The priority ceiling protocols were developed to minimize the priority inversion and blocking time.
Schedulability Criteria	*The schedulability test for **PCP** is:* $U = \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq i(2^{\frac{1}{i}} - 1)$, *where blocking figure B_i is the worst case computation time of the longest critical section of a task of lower priority than task i [54].*

on this is then built. Once a schedule is made, it cannot be modified online. *Static scheduling* is generally not recommended for dynamic systems. Applications like process control can benefit from this scheduling, where sensor data rates of all tasks are known before hand. There are no explicit *static scheduling* techniques except that a schedule is made to meet the *deadline* of the given application under known system configuration. Most often there is no notion of priority in static scheduling. Based on task arrival pattern, a time line is built and embedded into the program and no change in schedules are possible during execution [1].

2.1.4 Real-time Semantics And Tools

LIFE is the process $\alpha LIFE = \{beat\}$ *which can stop (die) at any time.*
$traces(LIFE) = beat^*$. [1]

Semantics is a powerful instrument for expressing behavior in a formal manner, so as to be universally understood and verified for correctness. Regarding the semantics of time and real-time systems, several models have developed. The most important ones are presented in this section, in a short overview.

Temporal Logic: In the narrowest sense, *temporal logic* comprises the design and study of specific systems for representing and reasoning about time. These enterprises may have both an applied and a theoretical side, the former consisting of designing a system (that is, making choices in the fields of ontology, syntax and semantics), formalizing temporal phenomena in it, and then putting it to work (perhaps through implementing it). On the theoretical side, one aims to prove formal properties of the system, such as *completeness* or *decidability.*

Temporal Logic is introduced as an extension to the classical propositional logic. The first basic idea underlying temporal logic is to address this issue by making pure logical valuations to be time-dependent. More precisely, one associates a separate valuation with each point of a given flow of time.

Def 1. *Let* $\mathcal{T} = (T, <)$ *be a flow of time. A valuation on* $\mathcal{T}$ *is a map* $\pi : (T \to (\Phi \to 0, 1))$. *Here* Φ *denotes the set of propositional variables. A model is a pair* $\mathcal{M} = (\mathcal{T}, \pi)$ *consisting of a flow of time and a valuation [55].*

Communicating Sequential Processes (CSP)/Timed-CSP: *Hoares CSP* [56] is an event based notation primarily aimed at describing the sequencing of behavior within a process and the synchronization of behavior (or communication) between processes. *Timed-CSP* extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronization [57], [58].

[1] http://www.comp.nus.edu.sg/ pat/csp-slide.pdf

A process is determined (specified) by what it can do (i.e. a process is defined by its behavior). The *perceived behavior* of a process will depend upon the *observer.* We shall be mainly concerned with specifying the interaction between a system and its environment (i.e. external, or visible, behavior). A process engages in *events.* Each *event*, in this context, is an atomic action. The *set of events* that a process can possibly engage in is the *alphabet* of the process. A *trace* is a finite sequence of events and therefore, any execution of the process will be one of these sequences.

Timed Automata (TA): A semantic foundation for real-time systems based on *TA* is being introduced in this subsection. This foundation is then further used to model systems and to define the formal semantics of *TA*.

A *timed input/output transition system (TIOTS)* is a *labeled transition system* where actions have been classified as inputs or outputs, and where dedicated delay labels model the progress of time. In this case, the set of positive real-numbers is used to model time. Below, the commonly used notation for *labeled transition systems* is extended to *TIOTS.*

Def* 2. *TIOTS: *It is assumed that a given set of actions A is partitioned into two disjointed sets of output actions A_{out} and input actions A_{in}. In addition, it is also assumed that there is a distinguished unobservable action $\tau \notin A$. The set $A \cup \{\tau\}$ is denoted by A_τ. A* TIOTS *S is a tuple (S, so, A_{in}, A_{out}, $\rightarrow$), where:*

- *S is a set of states, $s_0 \in S$,*
- *and $\rightarrow \subseteq S \times (A_\tau \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation satisfying the usual constraints of time determinism (if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$ then s'=s"), time additivity (if $s \xrightarrow{d_1} s'$ and $s' \xrightarrow{d_2} s''$ then $s \xrightarrow{d_1+d_2} s''$), and zero-delay (for all states $s \xrightarrow{0} s$). $d, d_1, d_2 \in \mathbb{R}_{\geq 0}$ and $\mathbb{R}_{\geq 0}$ denotes non negative real numbers.*
- *$s \overset{a}{\Rightarrow} s'$ iff $s \overset{\tau}{\rightarrow}^* \overset{a}{\rightarrow} \overset{\tau}{\rightarrow}^* s'$, where $a \in A$,*
- *$s \overset{d}{\Rightarrow} s'$ iff $s \overset{\tau}{\rightarrow}^* \xrightarrow{d_1} \overset{\tau}{\rightarrow}^* \xrightarrow{d_2} \cdots \overset{\tau}{\rightarrow}^* \xrightarrow{d_n} \overset{\tau}{\rightarrow}^* s'$, where $d = d_1 + d_2 + \cdots + d_n$.*

Def* 3. *Timed Trace: *An observable timed trace $\sigma \in (A \cup \mathbb{R}_{\geq 0})^*$ is of the form $\sigma = d_1 a_1 d_2 \cdots a_k d_k$. Observable timed traces TTr(s) of state s are defined as:*
$TTr(s) = \{\sigma \in (A \cup \mathbb{R}_{\geq 0})^* | s \overset{\sigma}{\Rightarrow}\}$. For a state s (and subset $S' \subseteq S$) and a timed trace σ, s After σ is the set of states that can be reached after σ:
s After $\sigma = \{s' | s \overset{\sigma}{\Rightarrow} s'\}$, S' After $\sigma = \bigcup_{s \in S'} s$ After σ

TA [92] is an expressive and popular formalism for modeling real-time systems. Essentially, a *TA* is an extended finite state machine equipped with a set of real-valued clock-variables that track the progress of time and that can guard when transitions are allowed.

Def 4. *Let consider X to be a set of $\mathbb{R}_{\geq 0}$-valued variables called clocks and let $\mathcal{G}(X)$ denote the set of guards on clocks, being conjunctions of constraints of the form $x \bowtie c$, where $\bowtie \in \{\leq, <, =, >, \geq\}$.*

Def 5. *Let $\mathcal{U}(X)$ denote the set of updates of clocks corresponding to sequences of statements of the form x:=c, where $x \in X$, and $c \in \mathbb{N}$.*

Def 6. *A **TA** over (A, X) is a tuple (L, l_0, I, E), where:*

- *L is a set of locations, $l_0 \in L$ is an initial location,*
- *$I : L \to \mathcal{G}(X)$ assigns invariants to locations, and*
- *E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times A_\tau \times \mathcal{U}(X) \times L$.*

It can be written $l \xrightarrow{g,\alpha,u} l'$ if $(l, g, \alpha, u, l') \in E$. The semantics of *TA* is defined in terms of a *TIOTS* over states of the form $s = (l, \bar{v})$, where l is a location and $\bar{v} \in \mathbb{R}_{\geq 0}^X$ is a clock valuation satisfying the invariant of l. Intuitively, a *TA* can either progress by executing an edge or by remaining in a location and letting time pass:

$$\frac{\forall d' \leq d. I_l(d')}{(l, \bar{v}) \xrightarrow{d} (l, \bar{v} + d)} \qquad \frac{l \xrightarrow{g,\alpha,u} l' \wedge g(\bar{v}) \wedge I_{l'}(\bar{v'}), \bar{v'} = u(\bar{v})}{(l, \bar{v}) \xrightarrow{\alpha} (l', \bar{v'})} \tag{2.1}$$

In delaying transitions, $(l, \bar{v}) \xrightarrow{d} (l, \bar{v} + d)$, the values of all clocks of the automation are incremented by the amount of the delay d, denoted $\bar{v} + d$. The automaton may delay in a location l as long as the invariant I_l for that location remains true. Discrete transitions $(l, \bar{v}) \xrightarrow{\alpha} (l', \bar{v'})$ correspond to execution of edges (l, g, α, u, l') for which the guard g is satisfied by $\bar{v}$, and for which the invariant of the target location $I_{l'}$, is satisfied by the updated closk valuation $\bar{v'}$. The target state's clock valuation $\bar{v'}$ is obtained by applying clock updates u on $\bar{v}$ [93].

A variety of tools for formal verification of real-time systems (e.g. **Process Analysis Toolkit(PAT)** [113], *Uppaal* [59], etc.) have been developed on top of the previously introduced semantic models. [12] provides a comprehensive state of the art of such frameworks, together with the presentation of additional semantic models.

Formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property. *Testing* is the process of analyzing a software to detect the differences between existing and required conditions (identify bugs) and to evaluate the features of the software. It can be stated that while *formal verification* reasons whether or not the analyzed software is suitable during the verification phase, *testing* experiments with the program. This makes *testing* a suitable technique for verification but also for validating a system as a final product.

Formal verification and *testing* are two different practices of software *V&V*. They are not mutually exclusive, but rather complementary. *Formal verification* is effective in proving the *correctness* of an application, from the early stages of specification design, but it has a medium efficiency in finding *bugs* at execution time. On the other side, *testing* is apt in finding *bugs*, although it cannot prove *correctness*. Also, when comparing expenses, it can be observed that *testing* is less costly than *formal verification*, specially with regards to the verification of complex systems. Those aspects can be compared at a glance in Table 2.7.

	Testing	***Formal Verification***
Finding Bugs	good	medium
Providing Correctness	–	good
Cost	small	high

Table 2.7: Testing vs. Formal Verification.

The focus of this thesis is on *testing* as a method of *V&V* for real-time and embedded systems. Nevertheless, some of the formalism presented in this section is used, in the context of the presented solution, as a means of specifying the semantics of the real-time test specification design.

In the following Section (2.2), the discussion about real-time systems verification is refined further, by identifying *black-box testing* as the main adopted technique and by presenting the *black-box testing concepts* in relation to real-time.

2.2 Testing For Real-time

Testing is the execution of the *SUT* in a controlled environment. It follows a prescribed procedure with the goal of measuring one or more quality characteristics of a product, such as functionality or performance. According to [100], *testing* is the primary software validation technique used by industry today and represents a fundamental step in any development process. It consists of applying a set of experiments to a *SUT*. There exist many types of testing with multiple aims, from checking correct functionality to measuring performance.

In this thesis, the aim is the validation of a real-time *SUT* with regard to a given specification. The targeted *SUT* is a final product of real-time industry, regarded as a *black-box* in the sense that we do not have knowledge about its internals (e.g., its state is not known). Thus, one can only rely on its observable input and output behaviour [101].

There are two basic classes of software testing, *black-box testing* and *white-box testing*, denoted by colors to depict the opacity of the testers of the code. With *black-box testing*, the software tester does not (or should not) have access to the source code itself. Alternatively, *white-box testing* focuses on the internal structure of the software code. In the language of *V&V*, *black-box testing* is often used for validation and *white-box testing* is often used for verification [11]. In the context of this thesis the focus is on *black-box testing.*

Timing of inputs & outputs. Designing a testing solution for real-time systems is not a trivial task. These are systems that operate in an environment with strict timing constraints. When testing a real-time system, it is not sufficient to check whether the *SUT* produces the correct outputs. It must also be checked that the *timing of the outputs* is correct. Moreover, the *timing of these outputs* depends on the *timing of the inputs.* In turn, the timing of applicable future inputs is determined by the outputs [101].

Validation of the *SUT* in its operating environment. A real-time embedded system interacts closely with its environment, which typically consists of the controlled physical equipment accessible via sensors and actuators, other computer based systems or digital devices accessible via communication networks using dedicated protocols, and human users. A major development task is to ensure that an embedded system works correctly in its real operating environment.

The goal of conformance testing is to check whether or not the behavior of the *SUT* is correct according to its specification, and under the assumptions about the behavior of the actual environment in which it is supposed to work. In general, only the correctness in this environment needs to be established, or it may be too costly or ineffective to achieve for the most general environment [99].

The Timeliness of the *SUT* implies the timeliness of the *TS*. Considering the strong interconnection between the *TS* and the subject of its testing, any *TS* should be designed with the *SUT* in mind. The properties of the *SUT* and the features that are to be tested are of the highest importance when building the *TS*. Knowing that the *timeliness* of the *SUT*, as a real-time application, is one of its most important properties, the *TS* should possess a correspondent timeliness as well.

Robustness of the *TS* is important. *Black-box testing* procedures consist of sending stimuli to the *SUT* and waiting for responses. Those responses should match to some *templates* that had been defined in the specification of the *SUT*. Based on wether there was a match or not, a *test verdict* is established. Invalid responses from the *SUT* should be also taken in consideration when designing a *TS* and thus, assuring the *robustness* property of the *TS*. Theoretically, the *TS* should be ready to manage any possible response of the *SUT*. *TS* should recognize failures from the *SUT* and handle them appropriately.

Designing the *TS* with regards to different time and content variations in the responses of the *SUT*. When talking about *functional tests*, *failure* means that the response from the *SUT* is not conforming to certain patterns defined in the specification of the *SUT*, or they don't respect their sequencing, or they fail to arrive. All these situations should be taken into consideration when designing the *TS*.

Adding time dimension to the testing process makes the problem more complicated. The messages and exchange of messages between the *TS* and the *SUT* should not only respect certain patterns and sequencing, but they also should be delivered and received at certain points in time. We are going to denote further incoming timed messages from the *SUT* to the *TS* as *events* and the outgoing messages (from *TS* to the *SUT*) as *stimuli*.

The communication between *TS* and the *SUT* must be real-time. Because the *SUT* is real-time, the communication between the *SUT* and the *TS* should also be a real-time communication. Therefore, by induction, the *TS* should be a real-time application as well. The application should be robust enough to handle a wide range of events, and responsive enough to handle them in time.

Responses from the *SUT* are regarded at the *TS* as events with strict time constrains.

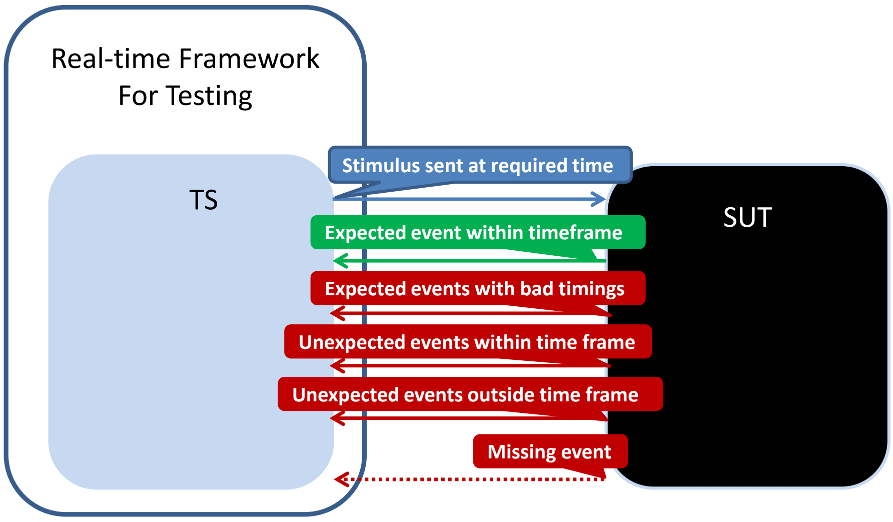

Real-time Framework For Testing
TS
SUT
Stimulus sent at required time
Expected event within timeframe
Expected events with bad timings
Unexpected events within time frame
Unexpected events outside time frame
Missing event

FIGURE 2.2: Black-box Testing For A Real-time System

The main classes of events, presented also in Figure 2.2, are:

- *Expected events that come in the expected time frame:* they indicate a good functionality of the *SUT*.
- *Expected events with bad timings:* the messages correspond to the patterns, they come in the right sequence, but they are either delayed, or prematurely received.
- *Unexpected events:* This category can be split into two subclasses:
  - *Unexpected events within time frame:* the messages are either originally correctly transmitted messages, which get corrupted through the medium, or are completely wrong messages, indicating a malfunction of the *SUT*. Although received in the correct time frame, these messages will not match the conformance templates and should indicate failure.
  - *Unexpected events outside time frame:* messages which are both incorrect and with bad timings. Those messages will match neither the conformance templates nor the timed requirements patterns and should indicate failure.

  These kind of faults might be dangerous when they arrive in high volumes; they might flood the *TS*, causing a slowing down of its processes, and therefore might cause missing deadlines. A robust and reliable *TS* should have protection mechanisms to avoid these situations.
- *Missing events:* this class indicates that no message was received from the *SUT* before the upper limit of the expectancy time had been reached. In order to avoid a situation when the *TS* hangs in a blocking state forever, this state should be exited after the time expires, and the failure should be then acknowledged.

Real-time Test System (*RTTS*). As previously stated, the *TS* used for testing a *real-time SUT (RTSUT)* must satisfy the timeliness property, and must provide real-time communication with the *SUT*. Therefore, *TS* can be regarded as a real-time application itself (*RTTS*), ready to handle the incoming events from the *SUT*.

The *RTTS* should posses a dynamic behavior, able to handle these events. Because some of the aforementioned events are time-constrained, a precise time cognizance, intrinsic to the *RTTS*, should be present. This timing awareness is given by an *RTTS*-internal clock, together with a correspondent mechanism for managing events.

Two main categories of events that influence the behavior of the *RTTS* can be distinguished:

Time events (or internal events) triggered by the inner clock. Represents an intrinsic mechanism of control.

Incoming events (or external events) triggered by incoming messages from *SUT*.

Test Components with real-time. Multitasking might be required for the handling of various events. This means that the *RTTS* might consist of multiple entities running in parallel. In the context of *black-box testing* those entities are the *test components*. In the context of testing for real-time, the timing aspects of *test components* become relevant. Mechanisms for saving time properties (e.g. the start or the end of execution of a test component), as well as an appropriate scheduler are both necessary for managing their execution. An appropriate scheduling algorithm must be the next step employed for this purpose. Due to the aim of this thesis, for designing a testing framework able to address a wide variety of real-time applications, a generic real-time scheduling technique should be adopted. Therefore, in this context, the *EDF* algorithm for scheduling would be a suitable choice [137], [138].

Aspects regarding the execution of the *RTTS*. As we have to consider unpredictable incoming events at unpredictable times, we should also provide a means for handling such events. The handling procedure implies the existence of asynchronous tasks. The messages should not wait too long in the queue of the receiving port. They should be treated immediately. Therefore, the tasks that handle these events should become run-able as soon as possible. It should therefore be possible for them to preempt the current task. An operating system that provides preemption is chosen as the basis for our *RTTS* implementation. In this context, a dynamic priority based schema can also enhance the flexibility of the real-time scheduler, allowing the *RTTS* to carry out more robust behavior in reaction to the *SUT*.

As the discussion, in the following, will always involve testing of a *RTSUT* using a *RTTS*, for simpleness of notation, they will be referred from now on, throughout the thesis, by their short acronyms: *SUT* and *TS*, respectively.

2.3 Summary

This chapter has introduced concepts and characteristics of real-time, with regards to applications, programming languages, operating systems, formal semantics and testing. Firstly, the characteristics and requirements of real-time applications were studied, and based on those, the main features of real-time programming were identified. This discussion evolved around four real-time programming languages: Ada, Real-Time Euclid, Real-Time Java and Real-time C/POSIX. The extensions to TTCN-3 for real-time, adopted in Chapter 4, are based on the language mechanisms presented here. The chapter continued with the review of real-time concepts and mechanism applicable to real-time operating systems. This presentation offers a lower level perspective of the real-time domain and would be useful at the design and implementation step for the testing framework (Chapters 5 and 7). A description of the main semantic models for the real-time followed, in order to provide the theoretical foundation needed to specify the semantics of the real-time concepts for TTCN-3 in Chapter 4. At the end of this chapter, various aspects of *black-box testing* with regard to real-time were discussed. *Black-box testing* was the V&V technique chosen to prove the quality of real-time and embedded applications.

Chapter 3

State Of The Art

Those who do not want to imitate anything, produce nothing.
Salvador Dali

This chapter will discuss by means of tools and approaches the current state of the art in real-time testing domain in general, and in particular, real-time testing with TTCN-3.

3.1 State Of The Art In Real-time Testing

The development of embedded systems is an essential industrial activity whose importance is increasing. Commonly, embedded software systems have to fulfil real-time requirements. The most important analytical method to assure the quality of real-time systems is testing. Testing is the only method which examines the actual run-time behavior of embedded software systems, based on an execution in the real application environment. Dynamic aspects like the duration of computations, the memory actually needed, or the synchronization of parallel processes are of major importance for the correct function of real-time systems, and therefore must be tested [66].

As previously stated, in Chapter 1, the goal of this thesis is: *To build a manufacturer-independent testing framework that combines functional test automation with real-time test automation by means of a standardized testing specification language.*

There are different approaches for automatizing of real-time systems testing processes and many frameworks were designed with this purpose in mind. Some of the most interesting current solutions are presented in this chapter, in order to provide a wider picture and the context in which the solution proposed in this thesis was developed. It will also provide the reader with points of comparison.

The comparison will evolve around the following main criteria:

1. *Real-time testing capabilities* - how adequate is the testing framework for testing of real-time systems (See Chapter 2)?
2. *Automation techniques* - the testing framework should enable test automation.
3. *Standardization* - the framework should be based on a standardized testing language.

4. *Testing technique or other type of* V&V *practice* - due to the fact that the target of testing here is represented by systems for which the software tester does not have access to the source code, a black-box testing approach is implied here. Nevertheless the testing frameworks under discussion employ a variety of *V&V* practices.

Secondary criteria, such as *test portability*, *test modularity*, *easiness-to-read of tests*, and *industrial usage* or *popularity* of the frameworks, are also going to be added as arguments into the discussion.

TABLE 3.1: Real-time Testing Frameworks. Part I.

Tool	***Test Specification's Language***	***Dedicated Test Language?***	***Standardized?***
TTG	*IF modeling language*	*no*	*no*
dSPACE	*Python*	*no*	*yes*
TestFarm Core	*PERL*	*no*	*no*
TALENT^{TM}	*Test Automation Language (TAL)*	*yes*	*no*
ADvantage	*Visual Basic C++*	*no*	*no*
UPAAL-TRON	*Upaal timed automata*	*no*	*no*

TTG framework for black-box testing, based on timed automata. In [101] *TTG framework* for *black-box conformance testing* of real-time systems is introduced. The specifications for the *SUT* are modeled as nondeterministic and partially-observable TA. TA model was chosen due to reasons of ease of modeling and expressiveness of specifications. The conformance relation is a timed extension of the input/output conformance relation of [67].

The solution is based on building a prototype test-generation tool, called *TTG*, on top of the *IF environment* [68]. The *IF modeling language* allows the user to specify systems consisting of many processes, which communicate through message passing or shared variables and includes features such as hierarchy, priorities, dynamic creation and complex data types. The *IF tool-suite* includes a simulator, a model checker and a connection to the un-timed test generator *TGV* [69]. *TTG* is implemented independently from *TGV*. It is written in *C++* and uses the basic libraries of *IF* for parsing and symbolic reachability of timed automata with deadlines. *TTG* takes the specification automaton as the main input, written in *IF language*, and can generate two types of tests: analog-clock tests which measure dense time precisely and digital-clock tests which measure time with a periodic clock. The tests are output in *IF language*.

dSPACE Framework. The framework developed by *dSPACE* is based on the idea that automated testing is performed by executing tests on a standard PC, which is interconnected to a HIL system. *HIL simulation* is a common practice as a black-box testing methodology for *ECUs*. There is a need for a concept to execute tests in real-time on the processor board of the *HIL simulator* synchronously with the execution of the simulated model. In [70] a concept is presented that meets such demands as running test cases with precise sample-time and high degree of reproducibility and deterministic time and functional behavior. Such tests can be written using *Phyton* as a standard object-oriented scripting language, and executed in real-time without the need to modify and recompile the real-time model.

TestFarm Core from BasilDev. TestFarm [71] is a software environment targeted at performing black-box testing of real-time embedded software that uses various standard peripherals, in a heavily asynchronous environment. Such systems, are, for example payment terminals, where software has to interact with human actions (keypad, LCD, buttons etc.) while communicating to other system components (supervision servers, smart-cards,) etc. The test scripts are gathered into a structured test tree, following the organization of the *SUT*'s features. After the *TS* is executed, this structure is respected in the generated test report. Test suites are constructed with test scripts written in the PERL language. The TestFarm Core automated testing system runs on Fedora Core Linux Distribution [71].

$TALENT^{TM}$ **from ReACT Technologies.** $TALENT^{TM}$ [72] is a highly modular, integrated and scalable automation platform specifically designed for test facility automation. It can be deployed on one computer, or spread across several computers. The integrated environment encompasses all aspects of the testing process, from test planning (non-real-time), through test execution (real-time), to data upload to corporate data bases. $TALENT^{TM}$ supports automated testing with the Test Automation

Table 3.2: Real-time Testing Frameworks. Part II.

Tool	***Real-time Testing Capabilities***	***Automation?***	***Testing Technique***
TTG	*yes*	*yes*	*black-box testing* *simulation* *model checking*
dSPACE	*yes*	*yes*	*HIL simulation*
TestFarm Core	*yes*	*yes*	*black-box testing*
$TALENT^{TM}$	*partially*	*yes*	*black-box testing*
ADvantage	*yes*	*yes*	*HIL simulation*
UPAAL-TRON	*yes*	*yes*	*black-box testing* *simultion* *monitoring*

Language (TAL) which is an extension of the Microsoft Visual Basic for Applications (VBA). TAL VBA extensions are for run-time data access, rule based exception handling classes, timers, facility control objects, and access to TALENT ActiveX components [72].

ADvantage simulation framework. In [73] a solution is provided for enterprise test automation, used to minimize the human effort required to complete all aspects of a test. Ideally, each test is encapsulated within a single entry point or file. *ADvantage* was developed for the purpose of performing enterprise test automation. Alternatively, many organizations develop their own Visual Basic C++ applications to provide enterprise test automation functionality that plugs into the *ADvantage* simulation framework. Advanced real-time simulation systems, such as the *ADvantage* simulation framework, emply a "simulation host/real-time target" architecture to provide a complete simulation system that is both fully deterministic and allows for asynchronous user interaction and display.

UPAAL-TRON. *UPPAAL-TRON* is a new tool for model based online black-box conformance testing of real-time embedded systems specified as timed automata. It represents a recent addition to the *UPPAAL* environment and is performing model-based black-box conformance testing of the real-time constraints of embedded systems. *TRON* is an online testing tool which means that it generates and executes, at the same time, tests event-by-event in real-time. *TRON* replaces the environment of the *SUT*. It performs two logical functions, stimulation and monitoring. Based on the timed sequence of input and output actions performed so far, it stimulates the *SUT* with input that is deemed relevant by the model. At the same time it monitors the outputs and checks the conformance of these against the behavior specified in the model. Thus, *TRON* implements a closed-loop testing system. It is important to note that we currently assume that inputs and outputs are discrete actions, and not continuously evolving [99].

Comparison of the real-time testing frameworks. Tables 3.1 and 3.2 give an overview of the features provided by the tools with regard to the comparison criteria discussed previously. It can be noticed that all of the enumerated tools boast automation and real-time (at least partially, e.g. $TALENT^{TM}$) black-box testing capabilities. *dSPACE*, *ADvantage* and *UPPAAL-TRON* support HIL simulation techniques and *TTG* and *UPPAAL-TRON* provide simulation and model checking additional to black-box testing methods.

All the listed frameworks, except $TALENT^{TM}$, use programming languages, which are not-testing-dedicated, to define their test specifications. Only two frameworks from the selection, *dSPACE* and *TestFarm Core*, rely on standardized programming languages for defining their test specifications – *Phyton* and *PERL* respectively. Nevertheless, both *Phyton* and *PERL* specifications are high-level, general purpose programming languages, and their usage for testing is somewhat strained.

Four tools from the selection – *dSPACE*, *TestFarm Core*, $TALENT^{TM}$, and *ADvantage* – represent commercial solution, used in the industry, while the remaining two – *TTG* and *UPPAAL-TRON* – are prototypes, resulting from research projects. The tool range presented here, was intended to be a sample that reflects accurately the current situation of real-time test development both in industry and research.

During the investigation of exiting solutions, none was found to cover all the criteria we established for our research. No testing framework for real-time was found, with focus on automation and basing its test development on a standardized testing specification language. In this thesis, the aim is to build such a framework.

3.2 Motivation For Choosing The TTCN-3 Language

Among the various languages and tools to design real-time tests, the Testing and Test Control Notation TTCN-3 technology has been selected to stay at the basis of the testing framework developed here. There are many benefits of using TTCN-3 – many of them are being presented in the following – but the main reasons to select this language are down to the fact that TTCN-3 is actually *the only standardized test technology enabling test automation*.

TTCN-3 was created at European Telecommunication Standards Institute (ETSI) more than 15 years ago and has successfully been adopted by the industry ever since. Covering aspects as protocol and service, component, integration and system testing, as well as testing of embedded and distributed system, TTCN-3 gained great popularity in application domains such as telecommunication, automotive (e.g. TTCN-3 as been choosen by AUTOSAR as their official test language for conformance tests [86]), technical medical equipment, etc.. TTCN-3 has also been adopted by the ITU-T [10].

TTCN-3 is a modern, powerful test language that supports all kinds of black-box testing. The TTCN-3 language was created due to the imperative necessity to have universally understood language syntax able to describe test behavior specifications. Its development was demanded by industry and science to obtain a single test notation for all black-box testing needs. In contrast to earlier test technologies, TTCN-3 encourages the use of a common methodology and style, which leads to a simpler maintenance of test suites and products.

Among several other good reasons for using TTCN-3, there are some mentioned in [24] and discussed in the following:

Standardization: Being an international, open and maintained standard with standardized interfaces, extensibility is built in. For designing real-time tests a language is required that is standardized and easily extendable with real-time specific concepts. TTCN-3 fits perfectly into this profile.

Technology Independent: Besides typical programming constructs, TTCN-3 contains all the important features to specify test procedures and test campaigns for all kinds of testing such as functional, conformance, inter-operability, or load tests. These test-specific features are unique compared to traditional scripting or programming languages, and above all, technology-independent. Test portability is ensured this way, and this is one of the features that is desired for the real-time testing framework developed here.

Abstract Test Implementation: TTCN-3 defines test cases on an abstract level. Thus, test developers can focus on developing tests instead of worrying about the implementation on a platform or operating system. This is very important in the case of real-time test design, which can gain in complexity at operating system level (e.g. threads scheduling). A TTCN-3 compiler translates the abstract tests into executable code. Errors can already be spotted and fixed at compilation level, avoiding them at runtime. The generated code can be reused by adapting it to any platform or technology, also enabling testing during any design stages. This increases the reusability factor of the test code and its suitability for regression testing.

Less Costs, More Efficiency Through Test Automation: TTCN-3 enables test automation which reduces manual interaction related to all test phases. It ensures efficient and systematic testing, saving both time and money. Test automation increases reliability and reduces the risk of human error. Tests can be run faster within a regression test, and they can be run over and over again with fewer overheads. Together with the property of being *standardized*, *automation* was one of the key-features looked after at the selection of the testing language.

A standardized language provides a lot of advantages to both test suite providers and users. Moreover, the use of a standard language reduces the cost of education and training, as a great amount of documentation and examples are available. It is obviously preferred to use the same languages for testing rather than learning different technologies for distinct test classes. Constant use and collaboration between TTCN-3 consumers ensures a uniform maintenance and development of the language.

3.2.1 Concepts Of TTCN-3

A brief presentation of the TTCN-3 testing language and of its basic concepts is provided in this section. The concepts presented here are also relevant for real-time testing.

TTCN-3 is a modular language and has a similar look and feel to a typical programming language. In addition to the typical programming constructs, it contains all the important features necessary to specify test procedures and campaigns for functional, conformance, interoperability, load and scalability tests like:

- *test verdicts*,
- *matching mechanisms* (to compare the reactions of the *SUT* with the expected range of values),
- *timer handling*,
- *distributed test components*,
- the ability to specify *encoding information*,

- *synchronous* and *asynchronous communication*,
- *monitoring*.

A TTCN-3 test specification consists of the following main parts:

1. *Test data and templates definition.*
2. *Function and test case definitions for test behavior.*
3. *Control definitions for the execution of test cases.*

Modules. The top-level building-block of TTCN-3 is a module. A TTCN-3 module has two parts the *module definition part*, and the *module control part*. The definition part contains the data defined by that module – *functions, test cases, components, types, templates* – which can be used everywhere in the module and can be imported from other modules. The *control part* is the main program of the module, which describes the execution sequence of the *test cases* or *functions*. It can access the *verdicts* delivered by *test cases* and, according to them, can decide the next steps of execution. The test behaviors in TTCN-3 are defined within *functions*, *altsteps* and *test cases*. The control part of a module may call any *test case* or *function* defined in the module to which it belongs.

Test System. A *test case* is executed by a *test system (TS)*. A *TS* consists of a set of interconnected *test components* with well-defined communication *ports* and an explicit *test system interface*, which defines the boundaries of the *TS*.

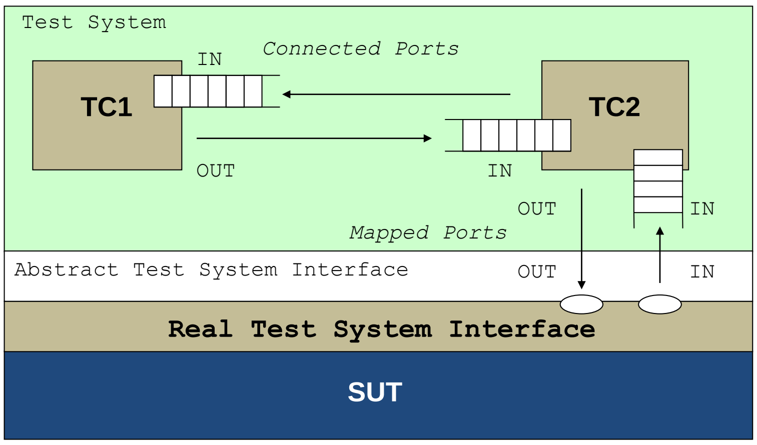


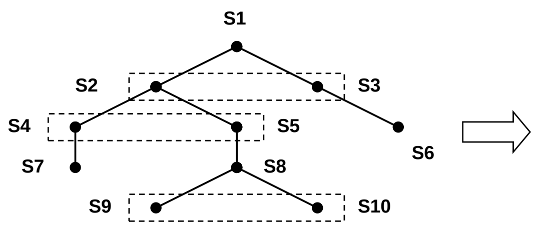
FIGURE 3.1: Conceptual View Of A TTCN-3 *TS* [2]

Within every *TS*, there is one *main test component (MTC)*. All other test components are called *parallel test components (PTCs)*, as shown in Figure 3.1. The *MTC* is created and started automatically at the beginning of each *test case execution*. A *test case* terminates when the *MTC* terminates. The behavior of the *MTC* is specified in the body of the *test case* definition. During the execution of a *test case*, *PTCs* can be created, started and stopped dynamically. A *test component* may stop itself or can be stopped by another *test component*.

Connected ports are used for the communication with other *test components*. If two ports are *connected*, the in-direction of one port is linked to the out-direction of the other, and vice versa. A *mapped* port is used for the communication with the *SUT*. The mapping of a port owned by a test component to a port in the *abstract TS interface* can be seen as pure name translation defining how communication streams should be referenced. TTCN-3 distinguishes between the *abstract* and the *real TS interface*. The abstract *TS* interface is modeled as a collection of ports that defines the abstract interface to the *SUT*. The *real TS interface* is the application specific part of a TTCN-3-based test environment. It implements the real interface of the *SUT*.

Test cases and test verdicts. *Test cases* define test behaviors, which have to be executed to check whether the *system under test* (*SUT*) passes the test or not. Like a module, a test case is considered to be a self-contained and complete specification that checks a *test purpose*. The result of a *test case execution* is a *test verdict*. A *test verdict* indicates the outcome of the test case execution, and can have one of the following values: *pass*, in the case of a successful execution, *fail* if the *SUT* proves to be non-conforming to the specification, *error* if the test case could not be completed due to technical problems at the *TS* and *inconc*, if the outcome of the test case execution was irrelevant.

Alternatives and snapshots. A special feature of the TTCN-3 semantics is the *snapshot*. *Snapshots* are needed for the branching of behavior due to the occurrence of *timeouts*, the termination of *test components* and the reception of *messages*, *procedure calls*, *procedure replies* or *exceptions*. In TTCN-3, the branching of behavior is defined by means of `alt` statements.



```
S1;
alt {[ ] S3; S6;
     [ ] S2;
        alt { [ ] S4; S7;
              [ ] S5; S8;
                    alt { [ ] S9;
                          [ ] S10;
                        }
              }
     }
```

FIGURE 3.2: Illustration Of Alternative Behavior In TTCN-3 [2]

An `alt` statement describes an ordered set of *alternatives*, i.e., an ordered set of alternative branches of behavior (Figure 3.2). Each alternative has a *guard*. A guard consists of several *preconditions*, which may refer to the values of variables, the status of timers, the contents of port queues and the identifiers of components, ports and timers. The same precondition can be used in different guards. An alternative becomes *executable*, if the corresponding guard is fulfilled. If several alternatives are executable, the first executable alternative in the list of alternatives will be executed. If no alternative becomes executable, the `alt` statement will be executed again.

The evaluation of several guards needs some time. During that time, preconditions may change dynamically. This will lead to inconsistent guard evaluations, if a precondition is verified several times in different guards. TTCN-3 avoids this problem by using *snapshots*. *Snapshots* are partial module states, which include all information necessary for the evaluation of alt statements.

A *snapshot* is a partial view of the current component that includes all information necessary to evaluate boolean conditions and alternatives: timeout events, stopped and killed components, pending messages, calls, replies and exceptions. A *snapshot* is taken, i.e., recorded, when entering an alternative. Each alternative is evaluated in the order of their appearance. For the verification of preconditions, only the information in the current *snapshot* is used. Thus, dynamic changes of preconditions do not influence the evaluation of guards.

Communication operations. Communication operations are important for the specification of test behaviors. TTCN-3 supports *message-based* and *procedure-based communication*. The communication operations can be grouped in two parts: *stimuli*, which send information to *SUT* and *responses*, which are used to describe the reaction of the *SUT*. The ports of TTCN-3 are used for handling the communication. As communication on ports is a very powerful construct, tailored to the needs of black-box testing, a sample of code is provided here, for a better understanding. Listing 3.1 shows an example on the usage of communication operations.

Timers. The central feature used in TTCN-3 for dealing with timing aspect of a given *SUT* is the *timer*. TTCN-3 has operations to start, stop, read or check if a timer is running, as shown in Listing 3.2.

Timers are declared, started and either stopped or ended with a timeout. When a timer times out, e.g. `aTimer.timeout`, the expired timer is placed in a timeout list. This list is checked when the next *snapshot* is taken. Timers may only exist in functions, test cases or in control part of a test suite, this is because there are no global timers (nor global data in TTCN-3). Timer is also a very important construct, especially with regard to

```
MyPort1.send(myValue);

MyPort2.send(myValue) to MyComponent1;

MyPort3.call(MyProc:{MyVar1}) to MyComponent2 {

        // in response the SUT may reply the value MyVar2
        [] MyPort3.getreply(MyProc:{MyVar2}) ()

        // or may generate an exception
        [] MyPort3.catch(MyProc, ExceptionOne) ()
}
```

Listing 3.1: An Example of Communication Operations in TTCN-3

```
 aTimer1.start;         aTimer2.start(2E-3);
//timer values of type float
aTimer1.stop;           all timer.stop;
//stopping inactive timer has no effect
var float aVar;         aVar := aTimer.read;
//assign to aVar time elapsed since aTimer started
if (aTimer.running) { }
//returns true/false if timer is running or not
```

Listing 3.2: Timers in TTCN-3

real-time testing. Therefore, for a better understanding of the concept, examples of timer usage in TTCN-3 are shown in Listing 3.3 [64].

More details and explanations about TTCN-3 can be found in the book [65].

3.3 State Of The Art Of Real-time Concepts For TTCN-3

Although TTCN-3 presents many benefits, that propose it as a good option for real-time testing as well, one should keep in mind that TTCN-3 was not originally conceived with real-time focus in mind. Therefore, there might be aspects of real-time testing (e.g. ensuring timeliness of the *TS*) for which TTCN-3 lacks the means to address.

In paper [78] some of the limitations of the existing treatment of time within TTCN-3 are illustrated as follows: the speed of the tester and the associated problems of the snapshot semantics and its impacts on accuracy of timing information; dealing with time critical testing information; the problems of time synchronisation of distributed test configurations. The paper proposes, as a work-around to the afore mentioned problems, some general guidelines for a more accurate measurement for real-time, using the actual capabilities of the language, within its boundaries.

Nevertheless, there already exist several approaches that have been proposed in order to extend TTCN-3 (and its earlier versions) for real-time and performance testing. Those enterprises are shortly summarized in the following, while a full picture of the relations

```
timer T1 := 10;
execute testCase1();
T1.start;
T1.timeout; // pause before executing next test case
execute testCase2();

execute (testCase3(), 5E-3) -> returnVal;
        // returnVal type verdictType = error if
        // result not returned in 5ms

P.call(X,5E-3); // can also put timeout value on procedure call
        { [ ] P.getreply(X);
        [ ] P.catch(timeout);
                {
                        verdict.set(fail);

                }
        }

while (T.running or x<10) {
        execute testCase4();
        x:= x + 1;
}
```

Listing 3.3: Timer Usage in TTCN-3

between them is presented in Figure 3.3. *PerfTTCN* and *RT-TTCN* are indicated as deprecated, because they rely on earlier versions of TTCN-3, and are, therefore, outdated.

PerfTTCN [74] extends TTCN (an earlier version of TTCN-3) with concepts for performance testing, such as: *performance test scenarios* for the description of test configurations, *traffic models* for the description of discrete and continuous streams of data, *measurement points* as special observation points, *measurement declarations* for the definition of metrics to be observed at measurement points, *performance constraints* to describe the performance conditions that should be met, *performance verdicts* for the judgement of test results. The *PerfTTCN* concepts are introduced mainly on a syntactical level by means of new TTCN tables. Their semantics are described in an informal manner and realized by a prototype.

RT-TTCN [75] is an extension of TTCN in order to test *hard real-time requirements*. On the syntactical level, *RT-TTCN* supports the annotation of TTCN statements with two *timestamps* for *earliest and latest execution times*. On the semantical level, the TTCN snapshot semantics has been refined and, in addition, *RT-TTCN* has been mapped onto *timed transition systems*.

In Table 3.3 can be visualized at a glance the extensions for TTCN.

Table 3.3: Extensions For TTCN.

Extension	***Target***	***Concepts***	***Syntax***	***Semantics***
PerfTTCN	*Performance Testing*	*- performance test scenarios* *-traffic models* *-measurement points* *-measurement declarations* *-performance constraints* *-performance verdicts*	*TTCN tables*	*informal*
RT-TTCN	*Hard Real-time Testing*	*- timestamps for earliest and latest execution times*	*TTCN tables*	*timed transition systems*

TimedTTCN-3 [76] is a real-time extension for TTCN-3, which covers most *PerfTTCN* and *RT-TTCN* features, while being more intuitive in its usage. Moreover, the *TimedTTCN-3* extensions are more unified than the other extensions by making full use of the expressiveness of TTCN-3. *TimedTTCN-3* introduces the following features:

- a new *test verdict* to judge real-time behavior.
- *absolute time*, as a means to measure time and to calculate durations. This is the reason for using the operation `now` at the current local time retrieval.
- `delay` provides the ability to postpone the execution of statements and the new statement `resume` provides the ability to delay the execution of a test component.
- *timed synchronization* for test components.
- the *timezones* concept, by which test components can be identified and can be synchronized in time.
- *online and offline evaluation* of real-time properties.

The real-time concepts introduced here represent a good basis for starting the design of a real-time test specification. However, they are only means of verifying the timeliness of the *SUT* and they do not guarantee the timeliness of the *TS*(e.g. that *TS* is able to stimulate and respond timely to the *SUT*). In order to impose a real-time execution upon the *TS*, further control mechanisms need to be added to the language.

ContinuousTTCN-3 [77] introduces basic concepts and means for handling continuous real world data in digital environments. Thus, TTCN-3 is enhanced, in this context, with concepts of stream-based ports, sampling, equation systems, and additional control flow structures to be able to express continuous behavior. In *ContinuousTTCN-3* time is also very important, and the problem of imprecise timers is mentioned. The concept of global time is taken from *TimedTTCN-3* and it is enhanced by the notion of *sampling* and *sampling time*.

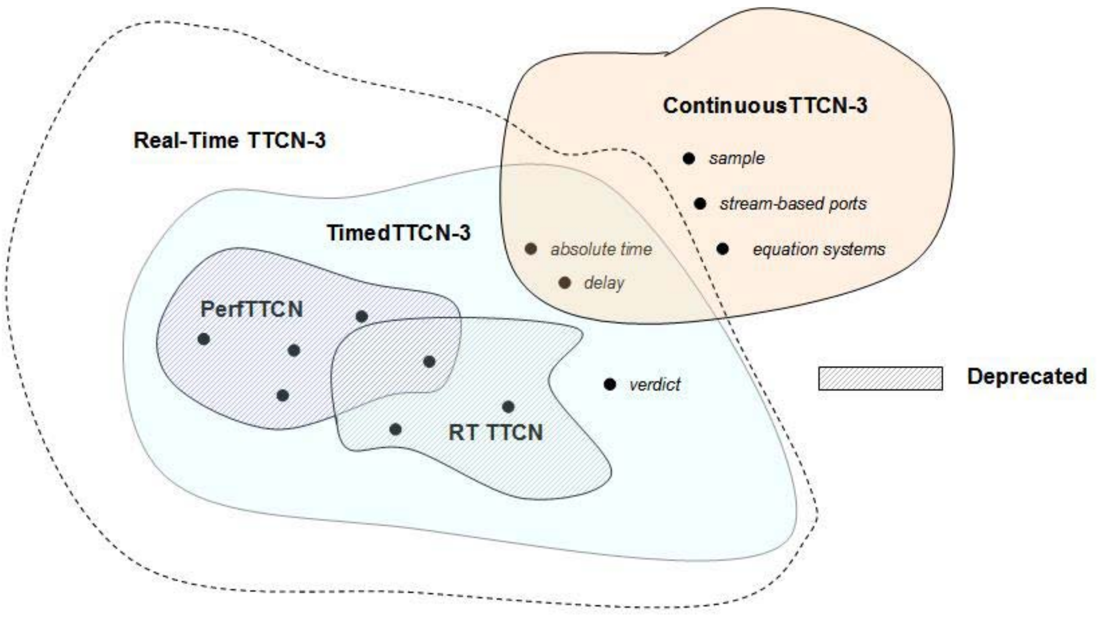


FIGURE 3.3: TTCN-3 Overview Of Proposed Extensions [3]

In Table 3.4 two older attempts of extending the TTCN-3 language with concepts for real-time and for continuous behaviour are displayed. More actual approaches towards real-time are provided in Tables 3.5 and 3.6. The solution presented in this thesis originates from these two latter extensions.

Real-time TTCN-3. The work presented in this thesis originates from the involvement of the author in the *TEMEA project* [79]. In the context of this project, a new paradigm and a new extension for real-time testing, based on the TTCN-3 notation was developed. The ambitious goal of this project was to use the experience of the past and to develop the language with new meaningful concepts, more powerful and more oriented towards the real-time world needs, than the attempts made in the past. The aims of the *TEMEA project* can be summarized in the following:

- Support for integrated testing of discrete and continuous behavior.
- Exchange of test definitions between different test- and simulation platforms (e.g. Model in the Loop (MIL) platforms, Software in the Loop (SIL) platforms, and Hardware in the Loop (HIL) platforms).

Table 3.4: Extensions For TTCN-3. Part I.

Extension	***Target***	***Concepts***	***Syntax***	***Semantics***
TimedTTCN-3	*Real-time Testing*	*-absolute time:* `now`, `delay`, `resume` *-time synchronization for test components:* `timezone` *-a new test verdict for online evaluation:* `conf` *-offline evaluation:* `log`, `logfile` *-logfile operations:*`first`, `next`, `previous`, `retrieve`	*BNF*	*informal*
ContinuousTTCN-3	*Handling Continuous Data In Digital Environments*	*-absolute time:* `now` *-stream-based ports:* `stream` *-sampling:* `@t` *-time partitions* *-additional control flow structures:* `carry`, `until`	*BNF*	*TPT state machines*

- Support over the entire process of software integration and hardware integration.
- Analysis of real-time and reliability requirements.
- Testing distributed components according to AUTOSAR architecture.
- Analysis of the quality of tests.

The basic concepts introduced by *TEMEA* can be organized in the following categories:

1. ***Representation of time:*** In order to ease the manipulation of time values, two new *abstract data types* are introduced: `datetime` for designating global time values and `timespan` for designating time distances or intervals between different points in time. `timespan` is used to represent the amount of time that has passed between events.

2. ***Measurement of time:*** The observation of time is directly connected to the reception and provisioning of messages, on the communication ports. A new construct is introduced for automatically registering the time value at which a receive or send event occurred. The saving is indicated by the redirect symbol → and the `timestamp` keyword. Additional keywords are introduced to save the time values of events, that are relevant for the *TS*: `testcasestart` returns the time value for when the test case execution started and `testcomponentstart` returns the time value for when the test component execution started. If needed, `now` operator can be used to interrogate the current value of the clock.

3. ***Control of application:*** The `@` operator is introduced to send certain messages at fixed points in time. It is associated with a `datetime` value, representing the point in time when the sending of a message must be performed. `suspend` operation might be used to postpone the current execution until a future time value.

4. ***Time verification:*** In order to verify whether or not certain messages were received in time, the `within` operator is introduced. The operator is associated with an interval of `datetime` values that represent the range for the allowed times of reception.

Because the work presented in this thesis began as a collaboration with the *TEMEA project*, the basic real-time concepts that are introduced in this document are based on the ones developed in the context of *TEMEA*, but are not limited to those (see Tables 3.7 and 3.8). Furthermore, the original concepts were further developed here by means of an accurate semantics (defined using TA) – one can see from Tables 3.5 and 3.6 that the original concepts display informal semantics – and by means of benchmarking their implementation on a concrete platform for a proof demonstration. There were also concepts developed in addition to those (e.g. `break...at` for `alt` instructions controlling the incoming communication, etc.). The concepts defined in this thesis were also successfully used in a real-time case study, for testing the functionality of an ECU, controlling an auto-door.

Some of the concepts listed above have gained concrete ground and become part of a new standard from ETSI in [80]. This standard is intended to be an extension for performance and real-time testing, and is regarded as an additional package to TTCN-3. TTCN-3 packages are intended to define additional TTCN-3 concepts, which are not mandatory as concepts in the TTCN-3 core language, but which are optional, as part of a package which is suited for dedicated applications and usages of TTCN-3.

To fulfil the requirements for testing real-time system, the following TTCN-3 core language extensions are being standardized in [80]:

1. A *TS* wide available clock, that allows the measurement of time during test case execution.
2. The current value of the *TS* clock can be accessed by means of the symbol `now`.
3. The requirements on the overall precision of the *TS* clock can be specified by means of the `stepsize` annotation.
4. The `wait` statement suspends the execution of a component until a given point in time. The time point is specified as a float value and relates to the internal clock.
5. Means to directly and precisely access the time points of the relevant interaction events between the *TS* and the *SUT*. Redirections for `receive`, `trigger`, `getcall`, `getreply`, and `catch` are extended by an optional clause consisting of the symbol → and the `timestamp` keyword.

(As previously mentioned, this thesis was developed in collaboration with *TEMEA* project.) The outcome of *TEMEA* remains at the basis of [80]. Therefore, it is natural that there are overlaps between those two extensions and the set of concepts considered here. One should observe Tables 3.7 and 3.8 for an overview of the concepts adopted here. From these tables one can see at a glance which of these concepts can be found in *TEMEA* and/or [80] and which concepts are new. If one extension is contained in either *TEMEA* or [80], this will be indicated by a ´✓ín the corresponding column.

The concepts that are the focus of this thesis are presented and discussed in detail in the following chapter 4.

3.4 Summary

This chapter provided the state of the art of the existing frameworks for real-time black-box testing, which also enable automation. A selection of tools, taken from both industry and academic environment, have been compared, based on criteria as: *the programming language that they use to define test specification*, *suitability of the specification language towards testing*, *standardization*, etc... The study revealed that there is no framework that relies on a standardized test specification language, and the goal of this thesis has been to build such a framework. The solution presented here was developed around the TTCN-3 standardized test specification language. There are plenty of reasons for choosing TTCN-3 and they have been presented in this chapter. Nevertheless, TTCN-3 was not defined with real-time focus in mind and therefore, it lacks some basic mechanism for handling real-time specific aspects of testing. In response to this need, different approaches have already been taken in order to extend the language with concepts for real-time. A state of the art of those extensions has been provided in this chapter. Some

TABLE 3.5: Extensions For TTCN-3. Part II.

Extension	***Concepts***		***Syntax***	***Semantics***
TEMEA	*Representation of time:*	*-abstract data types:* `datetime`, `timespan` *-time units:* `nanosec`, `microsec`, `millisec`, `sec`, `min`, `hour`, `day` *-conversion functions for absolute points in time*	*BNF*	*informal*
	Measurement of time:	*-timing for dedicated incoming communication events:* → `timestamp` *-timing for special events:* `testcasestart`, `testcomponentstart` *-current time:* `now`		
	Control of application:	*-control outgoing communication events:* `send@t` *-suspend execution:* `suspend`		
	Time verification:	*-verify incoming communication events:* `within` time interval		
ETSI Standard: TTCN-3 Performance and Real Time Testing	*Representation of time:*	-TS *wide available clock* *-time representation as* `float` *values* *-clock precision:* `stepsize` *annotation*	*BNF*	*informal*
	Measurement of time:	*-timing for dedicated incoming communication events:* → `timestamp` *-current time:* `now`		
	Control of application:	*-suspend execution:* `wait`		

Table 3.6: Extensions For TTCN-3. Part III.

Extension	***Concepts***		***Syntax***	***Semantics***
TEMEA	*Representation of time:*	*-abstract data types:* `datetime`, `timespan` *-time units:* `nanosec`, `microsec`, `millisec`, `sec`, `min`, `hour`, `day` *-conversion functions for absolute points in time*	*BNF*	*informal*
	Measurement of time:	*-timing for dedicated incoming communication events:* → `timestamp` *-timing for special events:* `testcasestart`, `testcomponentstart` *-current time:* `now`		
	Control of application:	*-control outgoing communication events:* `send@t` *-suspend execution:* `suspend`		
	Time verification:	*-verify incoming communication events:* `within` time interval		
ETSI Standard: TTCN-3 Performance and Real Time Testing	*Representation of time:*	-TS *wide available clock* *-time representation as* `float` *values* *-clock precision:* `stepsize` *annotation*	*BNF*	*informal*
	Measurement of time:	*-timing for dedicated incoming communication events:* → `timestamp` *-current time:* `now`		
	Control of application:	*-suspend execution:* `wait`		

Table 3.7: Real-time Extensions For TTCN-3. Our Approach. Part I.

Concepts		***TEMEA***	***ETSI Standard***	***Syntax***	***Semantics***
Representation of time:	*-abstract data types:* `datetime`, `timespan`	✓	–	*BNF*	*TA*
	-time representation as `float` *values*	–	✓		
	-time representation as `tick` *values*	–	–		
	-time units: `nanosec`, `microsec`, `millisec`, `sec`, `min`, `hour`, `day`	✓	–		
	-conversion functions for absolute points in time	✓	–		
Measurement of time:	*-current time:* `now`	✓	✓		
	-timing for special events: `testcasestart`, `testcomponentstart`	✓	–		
	-timing for special events: `testcomponentstop`	–	–		
	-timing for dedicated incoming communication events: $\rightarrow$ `timestamp`	✓	✓		
	-timing for dedicated outgoing communication events: $\rightarrow$ `timestamp`	–	–		

Table 3.8: Real-time Extensions For TTCN-3. Our Approach. Part II.

Concepts		***TEMEA***	***ETSI Standard***	***Syntax***	***Semantics***
Control of application:	*-suspend execution:* `wait`	✓	✓	*BNF*	*TA*
	-control on outgoing communication events: e.g. `send at`	✓	–		
	-control on incoming communication events: e.g. `alt..break at`	–	–		
	-control the starting and stoping of test components: e.g. `start at`*,* `stop at`	–	–		
Time verification:	*-verify incoming communication events:* `receive within` time interval	✓	–		
	-verify incoming communication events: `receive at/after/before` time point	–	–		
	-temporal predicates: `receive` temporal predicate	–	–		

of the most recent extensions have resulted in a new standard, containing additions to the language. The solution presented here originated from the author's collaboration in *TEMEA* project, a research project for ensuring quality of electronic components in the automotive industry. The concepts developed during this project remain at the basis of the new standard [80] standardized by ETSI and they also remain at the basis of the solution provided here.

Chapter 4

Our Approach Towards Real-time Testing

"Time must be the most paradoxical concept our minds have to deal with."
Confessions of St. Augustine

Testing involves any activity aimed at evaluating attributes or capabilities of programs or systems, and finally, determining if they meet all of the requirements. Black-box testing is purely based on the requirements of the *SUT* and it is mainly used for integration-, system- and acceptance-level testing. There exist several types of black-box testing which focus on different kinds of requirements or on different test goals, such as functional testing, conformance testing, interoperability testing, performance testing, scalability testing, and so forth. In the following sections, conformance testing for embedded real-time systems is considered and it is shown as an approach for systematically integrating timing properties into the testing process [4].

4.1 Real-time With TTCN-3 At Conceptual Level

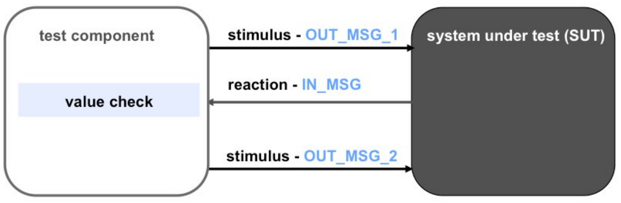


FIGURE 4.1: Simple Functional Black-box Test [4]

Figure 4.1 shows the setup for a simple functional black-box test that does not consider timing requirements. A *test component* is used to stimulate the *SUT*. The outputs of the *SUT* are captured by the *test component* and matched against predefined message templates. In this regard we distinguish between three different situations:

1. Message and template coincide, the tested requirement is considered to be fulfilled, and the test is continued.
2. Message and template do not coincide and the test fails.
3. No message arrives and the test fails.

Listing 4.1 shows a TTCN-3 implementation with respect for all cases.

```
timer t;
p_out.send(OUT_MSG);
t.start(TIMEOUT);
alt {
 /* The case when message and template coincide; */
 [] p_in.receive(IN_MSG) {p_out.send(OUT_MSG);}

 /* Message and template do not coincide and test fails; */
 [] p_in.receive {setverdict(fail);}

 /* No message arrives and after a while the test fails; */
 [] t.timeout {setverdict(fail);}}
setverdict(pass);
```

Listing 4.1: Simple black-box test with TTCN

In addition to the specific functional requirements defined by the specification, real-time systems also have to respect special requirements for timing [111]. Checking procedures based on the verification of the input/output values for the *SUT* are not sufficient in this situation. Since there are situations when the functional requirements are directly connected to the timing aspects, the degree of complexity increases. This means that certain functionalities must be accomplished within certain time intervals, with their starting or ending denoted by precisely defined time points or time spans, which also withstand a given tolerance. The tolerance is denoted in the following as $\Delta\varepsilon$.

In terms of testing, the above mentioned aspects might be verified with the use of components that are enhanced with the capability of checking whether or not the communication with the *SUT* respects the timings from the specification. If we consider that the communication with the *SUT* is message-based, then we have to ensure that a messages will be send, or received, at well defined points in time; those points in time are to be calculated with respect to preceding events (e.g. receiving of a message, start of the test case etc.); last but not least, we should be able to calculate and to compare time values with the required precision (e.g. micro- or even nano-seconds granularity). This level of granularity should be supported both at the test platform level and should be eased by the conceptual instruments as well.

An example test scenario with timing requirements is given in Figure 4.2. Regarding time, there are two critical sections in this test example: first, there is t_{max}, a time constraint for the *SUT* that indicates the interval in which the response to the first stimulus should be received by the *test component*; the second is t_{wait}, which indicates the time that should elapse at the *test component* side, between the receiving of the first response from the *SUT* and the sending of the second stimulus to the *SUT*.

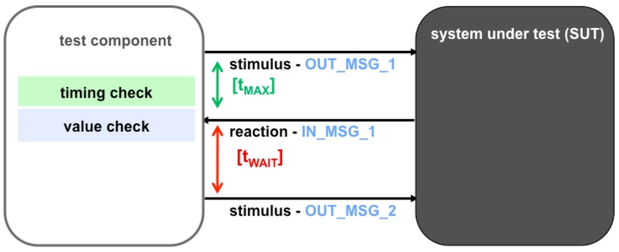


FIGURE 4.2: Black-box Test With Time Restrictions [4]

```
timer t;
p_out.send(OUT_MSG_1);

/* The timer is set initially to t_max*/
t.start(t_max);
alt{
    []p_in.receive(IN_MSG_1){
      /* After the test step is accomplished, the test component
       * should wait before sending another stimulus to the SUT;
       * the waiting period is set to t_wait;
       */
      t.start(t_wait);
      t.timeout;
      p_out.send(OUT_MSG_2);};
    []p_in.receive {t.stop;setverdict(fail)};

    /* This timer indicates that the interval for receiving
     * the first reaction from the SUT is t_max; after this
     * time expires, the test fails.
     */
    []t.timeout(){setverdict(fail)}
}
```

LISTING 4.2: Timing test with TTCN-3

A test logic for this example is comprised of traditional TTCN-3 in Listing 4.2 [4].

4.2 Why Is TTCN-3 Not Real-time

Before starting to introduce the approach that this thesis has adopted, we are going to discuss the limits of real-time testing with TTCN-3, as the language was initially conceived. TTCN-3 was originally designed for testing functional aspects of distributed solutions and not embedded real-time systems. By highlighting TTCN-3's limitations for real-time, the necessity for improvement and enhancement with new concepts was determined.

Obviously, traditional TTCN-3 already provides means to describe test cases that respect timing. Nevertheless, the solution denoted in Figure 4.2 has several disadvantages:

- The concept of a *timer* was not intended for suiting real-time properties, but conceived only for catching, typically mid- or long-term, *timeouts*. The specification of real-time properties using the concept of a *timer* is often clumsy and because of the non-real-time semantics of TTCN-3, not exact.

- *Timers* are not explicitly related with *events*; their semantics are implicitly defined by the place where the *timeout catcher* is positioned into the specification; there is no explicit correlation between the *sending/receiving* of a message and a *timeout.*

- Although the delaying of components for a certain period of time can be realized - as indicated in Listing 4.2, lines 12-13 - the semantics of this waiting action are not precise. What would happen, for example, if one or more instructions are inserted between lines 12-13 of the code sample? In this situation, the *timer* might generate a *timeout* long before the *timeout* instruction is reached. The timing when the *timeout* instruction is within reach of execution is unreliable. Another counterexample might be a situation, when the task realising the execution of the component is preempted in between the execution of the instructions from lines 12-13 for an undetermined amount of time. Again, by the time the *timeout* instruction gets to be processed, the real state of the *timer* could be long outdated.

- The use of a *timer* is affected by the TTCN-3 *snapshot semantics* (see Chapter 3, Section 3.2.1) and by the order in which `receive` and `timeout` statements are aligned in the `alt` statement. TTCN-3 in general makes no assumptions about the duration for taking and evaluating a *snapshot* which may vary, depending on different implementation solutions. Here it should be mentioned that TTCN-3 is defined as a testing specification, independent from its implementation. Theoretically, the time for taking the *snapshot* is considered to be null. This is obviously not achievable in practice, so the necessity emerges to define a specification that can be implemented in a manner that conduces to *limited* and *predictable execution times.*

- The measured time point that can be calculated based on the timer's value is in fact not the time point of message reception (i.e. the time a certain message has entered the input queue of the *TS*), but the time point of the evaluation of the queues by the test program (the time when the *snapshot* was taken). This kind of measurement approach is not exact.

- Additional time is consumed for the encoding and decoding of messages, as well as for the process of message matching. This time consumption is not taken into consideration by the *TS*. Furthermore, it is neither controllable nor assessable by the tester and introduces unpredictable inaccuracies for time measurements when using TTCN-3 timers.

```
var timespan stamp;

/*
 * Explicit mechanism for saving the exact time when the message
 * leaves the system.
 */
p_out.send(OUT_MSG_1) -> timestamp stamp;
alt{
 []p_in.receive(IN_MSG_TMPL)

   /* Explicit time restriction imposed at the message arrival.
    * Also the accurate time of message arrival is saved in a
    * variable for further usage.
    */
   not after (stamp+t_max) -> timestamp stamp{

     /*
      * Time constraint for imposing the sending of the message
      * at precise time point.
      */
     p_out.send(OUT_MSG_2) at (stamp+t_wait);}
 []p_in.receive(IN_MSG_TMPL){setverdict(fail);}
}
/*
 * Time constraint for the whole alternative; to prevent
 * blocking of the TS, in case when no message arrives.
 */
break at (stamp+2*t_max){setverdict(fail);}
```

Listing 4.3: Timed black-box test

- Last but not least, because the TTCN-3 is itself not real-time, the tester has no control over what happens between two instructions. If the test component that is currently running is preempted by another task, this will be transparent to the tester, leading to inaccuracies in test results. Furthermore, although these inaccuracies are due to the *TS*, the tester will not even be aware of them.

The concepts for RT-TTCN-3, that are introduced, are meant to overcome the limitations of traditional TTCN-3 mentioned above. Listing 4.3 informally introduces a TTCN-3 embedded implementation for the test task depicted in Figure 4.2. The timestamp operator (→ `timestamp`) writes the accurate point of transmission time of the message `OUT_MSG` into the variable `stamp`. This value can then be reused for the allocation of the `not after` operator, which is parameterized by a time stamp `stamp + t_max` that specifies the permitted arrival time of the message. The `at` operator in the following line specifies the exact point of message delivery. The `break at` restriction of the alt statement is used to set the *test verdict* to `fail` and thus prevent a deadlock in the case that no messages arrive in the indicated time. A more detailed specification of the concepts depicted above will be given in the following sections.

4.3 Requirements For Building A Reliable Real-time Test System

Real-time systems verification is required to prove not only the accuracy of computation, but also the fulfilment of the timing aspects of the system. In other words, we should ensure that the verified system (or the *SUT*) is time-predictable. This can be achieved only by using a *TS* that is time-predictable as well.

Timing properties of the system should be expressed from the early stages of system's specification and they should basically describe the *SUT*'s speed of reaction towards certain stimuli coming from the *SUT*'s environment. Timing requirements on the *SUT* could be specified at a different level of granularity, with a variation from hours, minutes, seconds down to nanoseconds. For precise testing of the timed reactions of the *SUT*, the *TS* itself should be equipped with an accurate clock, able to measure time at the desired degree of granularity. Together with the clock, an appropriate function for reading the value of the clock should be present, and some data type for representation should be used to save the values for further usage. When we deal with a *distributed TS*, the problem of measuring time can become more complex, since there is a need of *synchronizing* the clocks of each part of the system with the other parts, in order to keep consistency. The synchronization happens at certain time points that are common for the whole system and are referred to as *absolute*.When we deal with a *non-distributed TS* that rely on a single clock, we can measure time as being relative to certain events (e.g. the beginning of the test case) and we can only use *relative* time values. Thus, to reliably manipulate time in a *RTTS* we need to have: *a clock* that is capable of the desired precision, *a function to read the clock*, *data types for representing absolute and relative data values* capable of representing the value read from the clock.

Nevertheless, reading and saving time values of a clock would be not enough for making a system time-predictable. There should exist the possibility of associating durations with behavior. We should be able to *delay* the execution of certain parts of behavior for a specified period of time. Or we should be able to *resume* some blocking instructions (e.g. waiting incoming messages on ports) after a maximum time frame.

Black-box testing mainly involves an input/output based verification which is guided by a given specification of the *SUT*. Therefore, the timing aspects implied by a real-time *SUT* would consist of observing time at the communication ports and associating time with incoming/outgoing events. A means should be provided for associating such events with time data values, which could then be saved for further reference or calculation. Also, the communication on ports should be controllable at the tester's side. If a behavior associated with incoming communication takes too long, the *RTTS* should be able to interrupt that behavior and move forward. If an outgoing communication needs to be performed at a certain time point, the *RTTS* should be able to program that behavior in an appropriate way.

A test behavior might also be composed from several threads of execution, each responsible with verifying parts of the *SUT*'s timed or non-timed functionality. Conforming to the TTCN-3 specification, these threads are regarded as components. It is important to enhance the *RTTS* with the possibility of controlling the timed-behavior of the threads as well. This means that mechanisms for starting or ending the execution of the components at required time points should be provided. Also the possibility of saving the time point values when the components were started or ended should be provided.

The main aspects discussed here are summarized in Table 4.1.

–*clock* of high resolution
–function for *reading the clock*
–*data types* for representing *absolute* and *relative* time values
–*delay* the execution
–*time-stamping* incoming/outgoing communication events
–resume blocking instructions o incoming communication ports
–impose outgoing communication at specific times
–*time-stamping* start/stop of *test components*
–impose starting/stopping of *test components* at specific times

Table 4.1: Features Of A Reliable Real-time Test System

4.4 Real-time Extensions For TTCN-3

In the following, we shall discuss and introduce new instruments for dealing with real-time requirements in order to solve the previous presented problems (see Section 4.2). Carefully selected new additions to the language were developed as a consolidation of previous approaches [3, 76, 81]. The additions cover aspects like measuring timed reactions of the *SUT*, as well as controlling the timing of the *RTTS* itself. Our approach is based on the assumption that our *RTTS* should possess a time deterministic behavior with respect to the given testing tasks.

4.4.1 Data Types Suitable For Expressing Time Values

In order to ease the manipulation of time values, special data types are proposed for usage within a timed test specification described using TTCN-3. The discussion about the recommended data types, provides general guidelines on how time values should look like, how they should be interpreted, and how they can be converted from one data type to another. Nevertheless, the actual implementation of these time values depends on the coding techniques beneath the abstract level of TTCN-3. There are the options to integrate these data types as user-defined types and add them to the language in a separate module, or add them directly to the core grammar. Nevertheless, for the case in which the data types are to be incorporated into the grammar of TTCN-3, we are going to design the rules for integrating them among the other TTCN-3 types in Section A.1.

The introduced data types for handling time are:

1. `datetime` - with its mathematical domain $\mathbb{DT}$ - designates an ordered set of time points. These time points will uniquely specify the points in time when certain events occur or should occur. The `datatime` values presented in this thesis are ISO 8601:2004 [112] - compliant. They could be obtained through direct measurement, by reading some internal *clock* of the system. `datetime` values are intended to be used in distributed environments, where the synchronization between different components, which are running on different processors, is very important. As the aspects covered by this thesis focus mainly on a non-distributed test environment, the usages of `datetime` will not be emphasized here. It is, nevertheless, defined and presented, as it is considered useful for future studies in the area of distributed *TS* for distributed real-time applications.

2. `timespan` - with its mathematical domain $\mathbb{TS}$ - designates an ordered set of durations which express time distances or intervals between different points in time. It is used to represent the amount of time that has passed between certain events. `timspan` values are denoted using expressions that are constructed by positive *float* values multiplied by time units. Time units are represented by predefined constants (e.g. `min` for indicating minutes, or `sec` for indicating seconds, etc.). Because the durations expressed as `timestamps` are measured as time that passed from a given event, they are considered to be relative to that event, and are named *relative durations*. The `timespan` values can be expressed in minutes, seconds, milliseconds, etc. and they can be tuned towards the desired level of time granularity, down to the nanosecond. Because they are formed from a *float* value multiplied by a constant with the given semantics, they can be easily converted into float values and vice versa. Therefore, for the operators and functions that we are presenting in the following sections, the parameters and the returned values could be expressed either as `timespan` values or as `float` values. Conversion functions between the two types are also provided (see Appendix B). Nevertheless, the main purpose of `timespan` date type is to ease the usage of time duration, at different levels of granularity, in a human readable format.

3. `float` - with its mathematical representation $\mathbb{F}$ - indicates the domain of floating point numbers and is the basic type predefined in the standard [80]. The time

```
/* Declaration of datetime variables */
var float starttime, actualtime;
var datetime date;
var timespan ts_starttime, ts_actualtime;
var tick tck_starttime, tck_actualtime;

/* datetime value */
date:= 1982-02-22@22:10:50_0:0:0;

/* Usage of special operator which returns the point of time
 * when a certain event occurred, in this case the event is
 * the beginning of the test case execution. The value is in seconds.
 */
starttime:= testcasestart;

/* Read the value of the internal clock of the system, using
 * the special operator, now. The value is in seconds.
 */
actualtime:= now;

/* Conversion functions from float values to timespan values. */
ts_starttime:=seconds2timespan(starttime);
ts_actualtime:=seconds2timespan(actualtime);

/* Conversion functions from float values to tick values. */
tck_starttime:=seconds2ticks(starttime);
tck_actualtime:=seconds2ticks(actualtime);
```

Listing 4.4: Data types for saving time values

durations could be expressed not only as `timespan` values, but also as `float` values, depending on the preference of the test developer. When expressed by float values, the time durations will be indicated by default in seconds. Functions for conversion and format interchange between those two types are provided (see Appendix B). All the instructions for real-time that are going to be introduced, are recognizing `float` values as durations.

4. `tick` - with its mathematical representation $\mathbb{T}$icks - is equivalent with the set of positive integers and it indicates the value of the internal count unit of the CPU. The internal unit counts of the CPU are directly proportional with the frequency of the used CPU. This data type is introduced to enable a more precise evaluation of time events with a high degree of granularity, beginning already at the test specification level. The intended degree of time granularity for the test systems targeted in this thesis, is generally situated at the nanosecond level. Nevertheless, it is important to establish a direct correlation between the timing requirements of the *TS* with the time capabilities of the CPU. The capabilities of the CPU should be visible from the test specification.

```
var float distance, starttime;
var timespan maximum;

/* Calculating with time values */
starttime:= testcasestart;

distance:= now - starttime;

/* Initializing a timespan value */
maximum:= 200.0*millisec;

/* Comparison of timspan values */
if(seconds2timespan(distance) > maximum) {

    log("limit hurt")

}
```

Listing 4.5: Time related expressions

4.4.2 Special Operations Relaying On Time: `now`, `wait`, `testcasestart`, `testcomponentstart`, `testcomponentstop`

In order to give the tester access to *TS*-related events, some predefined operations are introduced. Together with the `now` instruction for reading the current time value from the contained clock, and the `wait` instruction for delaying the activity of the container component for a specified amount of time, we propose adding supplementary operators which are associated with important events in the life of the *TS*. Such events are the beginning of the *test case*, the starting and ending of *test components*.

Thus, the following operations are introduced in order to give the tester access to the time values associated with the aforementioned events. Unlike the `testcasestart` instruction, which refers to the current test case only, the `testcomponentstart` and `testcomponentstop` instructions could refer either to the current test component from within which this method is invoked, or to other test components which are declared in the *TS*. These instructions return the time values when the corresponding event has happened. The time values are expressed as `floats` which represent seconds and which can be easily transformed into `timespan` values, using the conversion functions from Appendix B.

The instructions defined in this section are shortly summarized, as follows:

1. `now` indicates the current time value.

2. `wait` forces the current component to wait for a given period of time.

3. `testcasestart` returns the time point when the test case execution started.

4. `testcomponentstart` returns the point in time when the test component execution started.

5. `testcomponentstop` returns the point in time when the test component execution stopped.

Time values can be used in arithmetic and boolean expressions (see Listing 4.5) respecting a few simple rules: the time values expressed as `float` indicate the number of seconds, and they can be easily converted to `timespan` values using conversion functions. The `timespan` domain, $\mathbb{TS}$, is an ordered set, and the expressions built using comparison operators, evaluates them to `boolean` values. `timespans` can be multiplied or divided by `floats` or `integers` and the result should be evaluated to `timespan`. The difference between two `datetime` values results in a `timespan` value. The syntax of numerical operators will be presented in Section A.1 from Appendix A. The overloading of numerical operators and the semantics for the timed expressions will be extensively presented in Appendix C, while Listing E.4 will present a few usage examples.

It can be observed that, unlike for the test components, we introduced a `testcasestart` operation for the test cases, but not a symmetrical `testcasestop` operation. In the context of a non-distributed *TS*, it was considered that there will be only one *test case* running at a time. Therefore, all the timed events happening during the running time of a *test case* can be tagged with times that are relative to the beginning of that specific *test case*. The `testcasestart` operation is particularly emphasizing this special type of event - the beginning of a *test case* - that would be very frequently used as a reference point by further events in the evolution of that *test case*'s behaviour. A `testcasestop` operation could serve either for the purpose of recording the time when a test case stopped, for logging purposes, or it can also serve for providing a point in time that could be used as reference by further behaviour. In the first case, for logging, the `now` operation can serve this purpose as well. Secondly, since the *TS* is non-distributed and all *test cases* are being executed sequentially, a `testcasestop` can serve as a reference point in time only for events that are contained in following *test cases*. Those events can, nevertheless, use the beginning of their own *test case*, as a better reference point, indicated by the `testcasestart` operation. The `testcasestop` operation would have been meaningful for a distributed system scenario, when different *test cases* could have been run in parallel and they could have used starting and ending times for synchronization among them. As the intention is to remain specific to the defined problem, and not to overload the TTCN-3 language with unnecessary constructs, the `testcasestop` operation has been left outside the scope of this thesis.

```
var float sendtime, receivetime;

/* Saving time value when the message left the test system */
p_out.send(MSG_OUT) -> timestamp sendtime;

/* Saving time value when the message entered the test system */
p_in.receive(MSG_IN) -> timestamp receivetime;
```

LISTING 4.6: Measurement of time

4.4.3 Measurement Of Time With `timestamp`

In RT-TTCN-3, the observation of time is directly connected to the reception and provisioning of messages at communication ports. We introduce a construct for automatically registering the time value at which a receive or send event has occurred. The saving is indicated by redirect symbol `->` and the `timestamp` keyword. The value is automatically stored as a `float` value in the indicated variable.

The `timestamp`-operator is available for message-based communications (i.e. for `send`, `receive`, and `trigger` statements) as well as for procedure-based communication[1](i.e. for `call`, `getcall`, `reply`, `getreply`, `raise`, and `catch` statements).

The observation of time with `timestamp` can be explicitly performed when a component is started or when a component is stopped. This way, it can substitute the usage of special operators `testcomponentstart` and `testcomponentstop` which implicitly return the same value.

Measurement of time with `timestamp` is discussed in detail in Sections 4.5.2, 4.5.3 and furthermore in Sections A.3, A.4, D.2, D.3, where a syntactical and semantical descriptions of concepts are provided, together with a few usage examples in E.3 and E.4.

4.4.4 Time Restrictions For Message Receival Using Time Predicates: `at`, `within`, `before`, `after`

In a *RTTS*, mechanisms should be provided not only for recording the times when messages were received, but also for imposing restrictions on the arrival times of the messages. For example, if one expected message is received outside the time frame specified in the requirements, it shouldn't be validated. Or even further, if a time frame for a message expires, we shouldn't spend additional time letting the component hang at a useless waiting operation on that port. Therefore, there is a need for stopping the effect of blocking operations such as `receive` when the allocated time frame for the messages expected on the receive port expires. In this manner, we avoid the blocking

[1]For more information about the differences between message-based communication and procedure-based communication and the corresponding TTCN-3 statements please refer to [2].

of the *TS* in the case in which the message expected from the *SUT* never arrives, and we can resume the execution with an appropriate behavior, designated to handle that situation.

In order to express the time frames for validating timing of messages, the following time predicates are defined:

1. `at` - indicating precise time points at which the messages should be received or at which a certain action should be performed. To that concern, the receival of a message on a port is regarded as an external event. The action to be performed by the *TS* is, for example, the canceling of a waiting operation on a port.

2. `within` - indicating a time frame delimited by two time points. The validated messages should be received in between the two time points associated with this predicate.

3. `after` - indicates that the validated messages are the ones received after a given time point.

4. `before` - indicates that the validated messages are the ones received before a given time point.

5 `not` - keyword placed before any of the above mentioned predicates, makes the expected time interval to be complementary to the time interval indicated by the predicate. The complement is calculated relative to the entire set of valid time points.

The time points that are used in conjunction with the aforementioned predicates are designated through `float` values expressing the passage of time in seconds. The `receive` instruction in addition with one or more of the aforementioned predicates, provides a means for validating the incoming messages with respect to time requirements. The semantics of this construct will be presented further in Section 4.5.4 and D.5, while its syntax will be explained in detail in Section A.6, and a usage example will be provided in Listing E.7.

For interrupting the waiting for messages on ports after a certain time frame expires, the `alt` together with `break at` construct should be used, as guards for the receiving branches. The functionality of this construct can be shortly explained in the following: if none of the `alt` branches are satisfied in a given time frame - and if the moment indicated by the parameter of the `at` predicate, associated with the `break` command, is reached - the waiting at the specified ports is interrupted and the behavior of the containing component is resumed by executing the block of instructions that follows the `break` command. The semantics of this construct will be presented further in Section 4.5.6 and D.7, while its syntax will be explained in detail in Section A.8, and a usage example will be provided in Listing E.9.

4.4.5 Inducing Events At Established Time Points Using Time Predicate `at`

Time predicate `at` can be additionally used to force the *TS* to perform an outgoing communication operation when some time point is reached. For example, it might be necessary, for real-time testing purposes, to stimulate the *SUT* at precise moments of time. This means that the `send` operation on the *TS* side should be performed at the indicated time. This can be achieved by extending the testing specification with the possibility of explicitly attaching time points to the `send` operation, time points that could be interpreted as a requirement addressed to the scheduler at the test execution. Further indications about semantics will provided in Section 4.5.5 and D.6 A.7. The syntactical structure will be given in Section A.7, while example of usage is provided in Listing E.8

In the same way that the action of sending a message should be allowed to be executed at a precise moment, the actions of starting and stoping of the components should also be allowed to be executed at precise times; or, rephrasing, it can be said that *test components* should be allowed to be *time triggered*. This is due to the fact that different *test components* are responsible for certifying different parts of the *SUT*'s timed functionality. Therefore, there is the need for the components to be able to be started at a precise moment in time, exactly when their intervention is desired. Otherwise, they might lose the grip on the timing aspects in relation with the *SUT*, and their estimations would not be valid.

We can achieve the launching of the `start` and `stop` of components at precise time points in combination with the `at` predicate. The approach is similar to the one used for imposing the sending of messages at specified times. Sections 4.5.7, D.8 and A.9 should be consulted for further explanation. Some usage examples are also presented in E.10.

4.5 Semantical Definitions Of The Real-time Extensions For TTCN-3 Using Timed Automata

The semantics for the new concepts will be presented in the following context. We consider our *TS* to be one type of timed automata (see Section 2.1.4) described by the set: $\mathcal{TS} = \{\mathcal{L}, \mathcal{X}, \mathcal{A}, \mathcal{G}, \mathcal{I}, \mathcal{U}, \mathcal{E}\}$, where:

$\mathcal{L}$ is a set of location or states from which we use a subset to define the semantics of the newly introduced concepts. This subset is

$\mathcal{S} =$
$\{S_{now}, S_{clock}, S_{wait}, S_{next}, S_{error}, S_{alt}, S_{break}, S_{wait\_alt}, S_{alt\_stop}\} \bigcup$
$\{S^i_{tc\_start} | i = 1..n_{testcases}\} \bigcup \{S^i_{comp\_start}, S^i_{comp\_stop}, S^i_{wait\_compstart}, S^i_{wait\_compstartstop},$
$S^i_{wait\_compstop}, S^i_{wait\_compstop\_stop} | i = 1..n_{comp}\} \bigcup \{S^i_{receive}, S^i_{rcv\_timestamp}, S^i_{start\_match},$
$S^i_{match\_wait}, S^{ij}_{match}, S^{ij}_{match\_time}, S^i_{match\_stop}, S^i_{send}, S^i_{send\_timestamp}, S^i_{wait\_send},$
$S^i_{wait\_stop} | i = 1..n_{ports}, j = 1..n_{tmpl_i}\} \bigcup \{S^k_{brunch} | k = 1..n_{brunches_{alt}}\}$

$\mathcal{X} = \mathcal{C}\mathsf{locks} \cup \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}$

- $\mathcal{C}\mathsf{locks}$ is considered to be a set of $\mathbb{R}^{>0}$-valued variables called clocks, where C_0 is the general clock of the system, visible from every state and which increments its value at fixed intervals; the general clock of the system cannot be reset or its value changed; all the other clocks are available to be used and initialized from any state; they are useful for calculating relative timings, such as measuring the time spent in a S_{wait} state, for example.
- $\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}$ is the list of all variables from the $\mathcal{TS}$.
- $\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} \supset \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}' = \mathcal{M}\mathsf{essages} \times \mathcal{T}\mathsf{imestamps}$, where:
  - $\mathcal{M}\mathsf{essages}$ represents the set of all messages that enter or leave the system at runtime.
  - $\mathcal{T}\mathsf{imestamps} = \{timestamp | timestamp \in \mathbb{R}^{>0}\}$ represents the set of timestamps for the messages that enter or leave the system at runtime. The timestamp variable represents time values in seconds.
  - $\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}'$ is organized into queues in the following way:
    $\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}' = \bigcup\limits_{i=1}^{n} queue_i = \bigcup\limits_{i=1}^{n}\bigcup\limits_{j=1}^{m_i} \{(msg_{ij}, timestamp_{ij}) | msg_{ij} \in \mathcal{M}\mathsf{essages},$
    $timestamp_{ij} \in \mathcal{T}\mathsf{imestamps}\}$
- $\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} \supset \mathcal{C}\mathsf{omponents}$, where $\mathcal{C}\mathsf{omponents} = \{comp_i | i \in \mathbb{N}^+, i = 1..n_{comp}\}$ is the set of all components that were created in the $\mathcal{TS}$.

$\mathcal{A} = \mathcal{C}\mathsf{hans}(\mathcal{A}_{in}) \cup \mathcal{C}\mathsf{hans}(\mathcal{A}_{out}) \cup \mathcal{A}'; \mathcal{A}_\tau = \mathcal{A} \cup \{\tau\}$ We assume a given set of actions $\mathcal{A}$, mainly partitioned into three disjoint sets of output actions:

- $\mathcal{C}\mathsf{hans}(\mathcal{A}_{in}) = \bigcup\limits_{i=1}^{n} Ch_i({\mathcal{A}_{in}}^i) = \bigcup\limits_{i=1}^{n} \{ch_i(a?) | a \in {\mathcal{A}_{in}}^i\}$, where ch_i are channels attached with communication ports of the $\mathcal{TS}$, n is a natural number representing the number of ports, and $Ch_i({\mathcal{A}_{in}}^i)$ is the set of input events that can enter on that port into the $\mathcal{TS}$.
- $\mathcal{C}\mathsf{hans}(\mathcal{A}_{out}) = \bigcup\limits_{i=1}^{n} Ch_i({\mathcal{A}_{out}}^i) = \bigcup\limits_{i=1}^{n} \{ch_i(a!) | a \in {\mathcal{A}_{out}}^i\}$, where ch_i are channels attached with communication ports of the $\mathcal{TS}$, n is a natural number representing the number of ports, and $Ch_i({\mathcal{A}_{out}}^i)$ is the set of output events that can leave the $\mathcal{TS}$ through the ith port.

- $\mathcal{A}'$ = is the set of special internal events used for synchronizing different parts of the $\mathcal{TS}$ which are running in parallel. The used set of internal events is: $\mathcal{A}' = \{tick!, tick?, now!, now?, break!, break?\} \bigcup \{received_i!, received_i?, queue_i!, queue_i?, received_{ij}!, received_{ij}?, send_i!, send_i?, stop\_send_i!, stop\_send_i? | i = 1..n_{ports}, j = 1..n_{tmpl_i}\} \bigcup \{start_i!, start_i?, stop_i!, stop_i?, stop\_comp\_start_i!, stop\_comp\_start_i?, stop\_comp\_stop_i!, stop\_comp\_stop_i? | i = 1..n_{comp}\}$
- In addition it is assumed that there is a distinguishable unobservable action $\tau \notin \mathcal{A}$

$\mathcal{G} = \mathcal{G}\mathsf{uards}(\mathcal{C}\mathsf{locks}) \cup \mathcal{T}\mathsf{emplates}(\mathcal{M}\mathsf{essages})$, where:

- $\mathcal{G}\mathsf{uards}(\mathcal{C}\mathsf{locks})$ denote the set of guards on clocks, being conjunctions of constraints of the form $c \bowtie t$, where $\bowtie \in \{\leq, <, ==, >, \geq\}, c \in \mathcal{C}\mathsf{locks}$ and $t \in \mathbb{R}^{>0}$
- $\mathcal{T}\mathsf{emplates}(\mathcal{M}\mathsf{essages})$ is a set of constraints on messages which splits the message set into equivalence classes of the form $EqMsgs = \{msg | msg \approx Tmpl, msg \in \mathcal{M}\mathsf{essages}\}$ and $Tmpl \in \mathcal{T}\mathsf{emplates}(\mathcal{M}\mathsf{essages})$. $\mathcal{T}\mathsf{emplates}(\mathcal{M}\mathsf{essages})$ is the set of all template applicable in the $\mathcal{TS}$.

$\mathcal{I} : \mathcal{L} \to \mathcal{G}\mathsf{uards}(\mathcal{C}\mathsf{locks})$ assigns invariants to locations.

$\mathcal{U}(\mathcal{X}) = \mathcal{U}(\mathcal{C}\mathsf{locks} \cup \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}) = \mathcal{U}(\mathcal{C}\mathsf{locks}) \cup \mathcal{U}(\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist})$ is the set of updates of clocks corresponding to sequences of statements of the form $c := t$, where $c \in \mathcal{C}\mathsf{locks}$ and $t \in \mathbb{R}^{>0}$ represents the time in seconds.

$\mathcal{E}$ is a set of edges such that $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{G} \times \mathcal{A}_\tau \times \mathcal{U}(\mathcal{X}) \times \mathcal{L}$

$\mathcal{Q} : \mathcal{C}\mathsf{hans} \to \mathcal{Q}\mathsf{ueues}$ is a bijective function which assigns queues to channels. For each channel ch_i there is a queue $queue_i$ associated with it so that $Q(ch_i) = queue_i$, where $i \in \mathbb{N}, i = 1..n$ with n designating the number of channels used by the $\mathcal{TS}$.

We consider $\mathcal{P}art$ to be the set of all partitions over the set $\mathcal{G}\mathsf{uards} \times \mathcal{T}\mathsf{emplates}$. The function $\mathcal{R} : \mathcal{Q}\mathsf{ueues} \to \bigcup_{\mathcal{P} \in \mathcal{P}art} \mathcal{P}$ assigns a set of time restrictions and message templates to each incoming queue. $\mathcal{R}(Queue_i) = \bigcup_{j=1}^{n_{tmpl_i}} g_i \times Tmpl_i$, with $g_i \in \mathcal{G}\mathsf{uards}, Tmpl \in \mathcal{T}\mathsf{emplates}, i, j \in \mathbb{N}$ and l_i is the number of templates associated with the queue.

We consider $\mathcal{B}\mathsf{runches}_{\mathcal{A}\mathsf{lt}}$ to be a subset of branches associated with an alternative. Then the function $\mathcal{B} : \mathcal{Q}\mathsf{ueues} \times \mathcal{G}\mathsf{uards} \times \mathcal{T}\mathsf{emplates} \to \mathcal{B}\mathsf{runches}_{\mathcal{A}\mathsf{lt}}$ represents a bijective relation which associate a queue with time guards and a template to a branch of the alternative. $\mathcal{B}(queue_i, TP_{ij}, Tmpl_{ij}) = branch_k$, where $i = 1..n_{ports}, j = 1..n_{tmpl_i}, k = 1..|\mathcal{B}\mathsf{runches}_{\mathcal{A}\mathsf{lt}}|$.

In the following we consider that the test system $\mathcal{TS}$ is a timed automata with the above presented structure. The semantics of the newly introduced instructions are represented accordingly, as timed automata that constitute subsections of the test system whose behavior they are forming. In this context, the $\mathcal{TS}$ would be a composition of those smaller time automata which may run in parallel.

4.5.1 Semantics Of Special Operations Relaying On Time: `now`, `wait`, `testcasestart`, `testcomponentstart`, `testcomponentstop`

At the core of a timed $\mathcal{TS}$ there is the `clock` which keeps track of time from the beginning of $\mathcal{TS}$ execution. Figure 4.3 presents the time automata associated with the global clock. The clock is considered to be periodic, with the period δt, this being a characteristic of the used processor, $\delta t = \frac{1}{f}$, where f is the frequency of the processor; $t, f \in \mathbb{R}^+$.

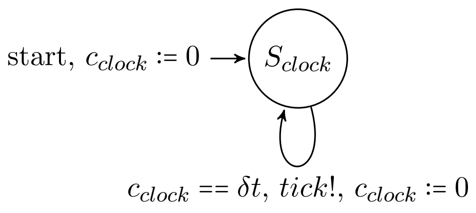


FIGURE 4.3: Logical Clock.

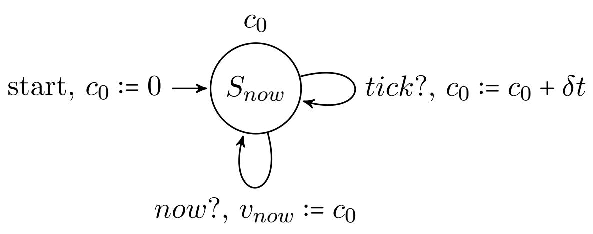


FIGURE 4.4: Now State.

Complementary to the timed automata for the logical clock, there is also the timed automata `now` which gives the total time from the beginning of $\mathcal{TS}$ execution, until the current moment when the time is requested (Figure 4.4).

When the period of time characteristic to the `clock` functionality expires, the `clock` automata emits a `tick!` signal. This signal is used for synchronization between the two automata. When received by the `now` automata, the local clock variable from the S_{now} state increases its value with the value of the period δt.

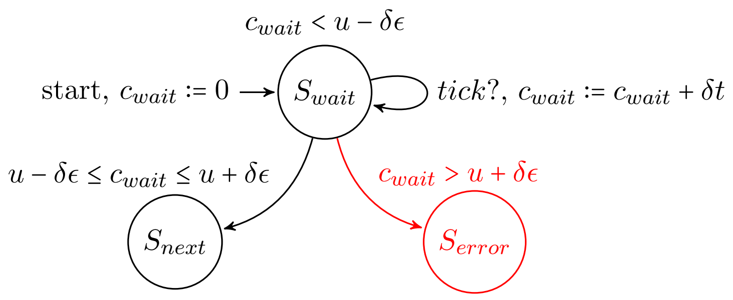

$c_{wait} < u - \delta\epsilon$
start, $c_{wait} := 0$
S_{wait}
$tick?$, $c_{wait} := c_{wait} + \delta t$
$u - \delta\epsilon \leq c_{wait} \leq u + \delta\epsilon$
$c_{wait} > u + \delta\epsilon$
S_{next}
S_{error}

FIGURE 4.5: Wait State.

The `clock` and `now` automata are going to be active for the whole life of the $\mathcal{TS}$. The `now` automata uses the `now?` signal for synchronization with other automata existing in the system. When some other automata needs to access the current time value, it will emit a complementary `now!` signal. This is intercepted by the `now` automata. Every time this signal is generated and intercepted, the value of the local clock c_0 is copied in the global variable v_{now}. This can be then accessed in the automata that has requested the current time.

A trace for the `clock` automata would look like: $(\delta t\ tick!)^*$, while a trace for the `now` automata might look like: $(tick?^n now?^m)^*$, where $n, m \in \mathbb{N}$ are naturals.

Figure 4.5 presents the timed automata associated with the `wait` instruction. It can be observed that the timed automata is approximately similar to the `now` automata, with the difference evident when the local clock, c_{wait}, reaches a first threshold but does not overpass a second threshold, there will be a transition to state S_{next}. If the second threshold is stepped over, then the transition will lead to an error state. The state S_{wait} has also the invariant $c_{wait} < u - \delta\epsilon$ associated with it. The states S_{next} and S_{error} represent generic states, used to designate the transition to other parts from the $\mathcal{TS}$, possibly represented by other timed automata.

S_{error} represents a final state indicating time inaccuracy of the `wait` instruction. This type of error should be signalized because otherwise the system would lose its propriety of being time deterministic.

A trace for the `wait` automata would look like $(tick?)^n t_{wait}$, where t_{wait} might be $t_{wait} \in [(u - \delta\epsilon), (u + \delta\epsilon)]$, or $t_{wait} > (u + \delta\epsilon)$.

The following automata represented in Figures 4.6, 4.7 and 4.8 illustrate the process of time stamping at the beginning of a test case, at the beginning of the execution of a test component, and at the end of the execution of a test component, respectively. The `next` state is not reached before interrogating the `now` automata for the current time. The current time is then stored in a variable, which is associated either with the beginning of the test case, or with the beginning or end of the test component. If we assume that there are n_{comp} test components in the current test case, then there will be $2 * n_{comp}$ variables for registering starting and ending time of each component.

As we assume that our $\mathcal{TS}$ runs on a single machine, we consider a sequential execution of the test cases and we keep a global variable for recording the starting point of the current test case.

4.5.2 Semantics For `receive` With `timestamp`

The timed automata from Figure 4.9 in composition with the timed automata from Figure 4.10 illustrate the `receive` mechanism with automate time stamping at the message arrival. The two automata should not be regarded separately, but rather in

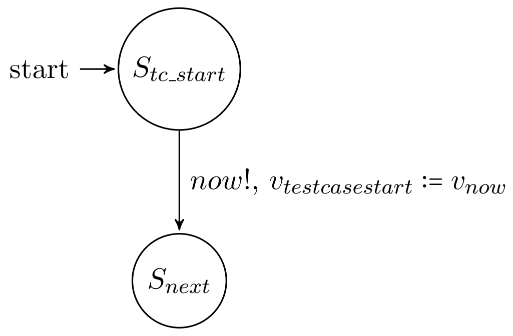


FIGURE 4.6: Testcasestart.

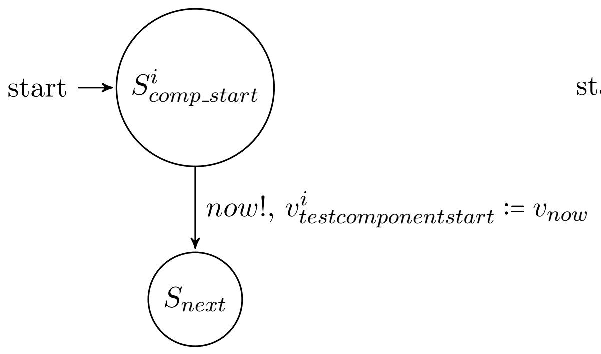


FIGURE 4.7: Testcomponentstart.

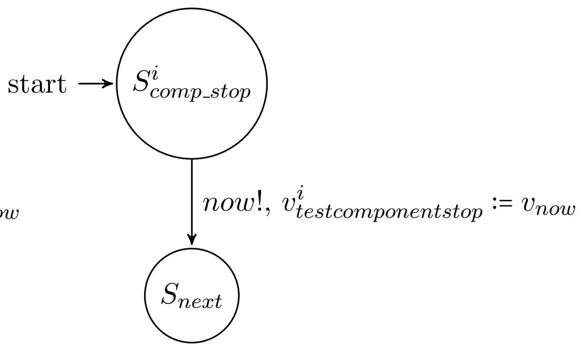


FIGURE 4.8: Testcomponentstop.

relationship to each other. The functionality is split between these two automata so as to increase precision of the time stamp value. The `receive` automata presented in Figure 4.9 has a functionality that should be simpler and therefore faster than the `match` automata presented in Figure 4.10.

The `receive` automata is triggered by the receival of a message on the input channel. When this happens, some basic actions are performed such as extracting the message and saving the current time. Then the signal $queue_i!$ is emitted for waking up the `match` automata which furthers the task of verifying whether or not the freshly arrived message is conforming to the template associated with that port. The operations performed by `receive` automata should be fast enough (and executed within predictable time bounds), and after they are accomplished, the `receive` automata is back to the $S_{receive}$ state, where it is free to receive other incoming messages, while the `match` automata may continue performing checking operations, which are usually much more time consuming and also very hard to predict, due to different lengths of messages and templates.

In the presented context, $ch_i(e?)$ means that one input event is expected on channel ch_i. The channels are associated to ports and are indexed according to the order in which the ports are used inside a test case. If there are n_{ports} ports, then $i = 1..n_{ports}$. $msg_{ij} = input(ch_i)$ means that when a message arrives on the channel ch_i it is extracted and saved in the variable msg_{ij}, where j indicates that this is the j-th message received

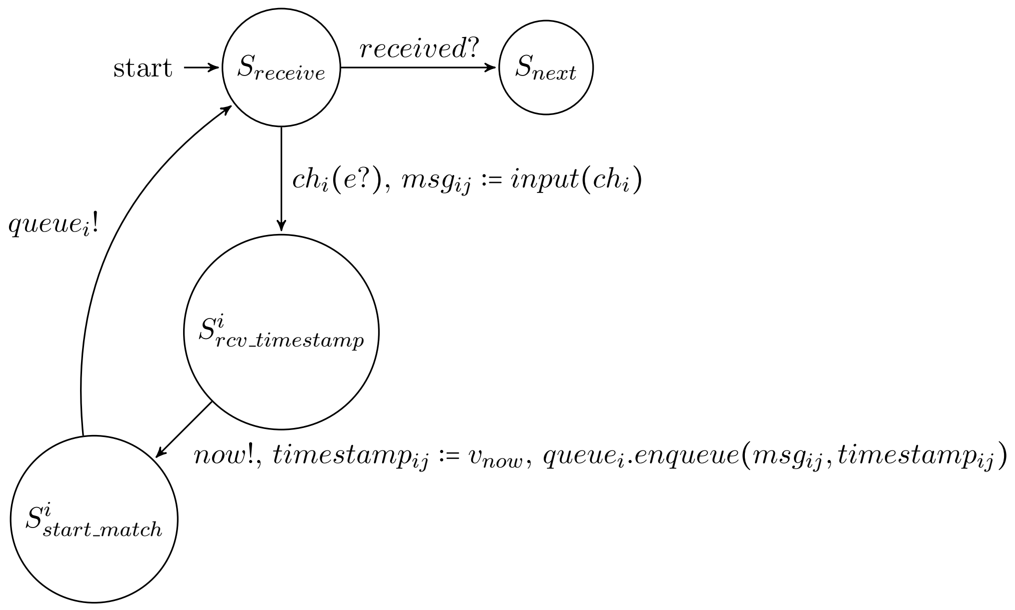

start
$S_{receive}$
$received?$
S_{next}
$ch_i(e?)$, $msg_{ij} := input(ch_i)$
$queue_i!$
$S^i_{rcv\_timestamp}$
$now!$, $timestamp_{ij} := v_{now}$, $queue_i.enqueue(msg_{ij}, timestamp_{ij})$
$S^i_{start\_match}$

FIGURE 4.9: Receive Automaton.

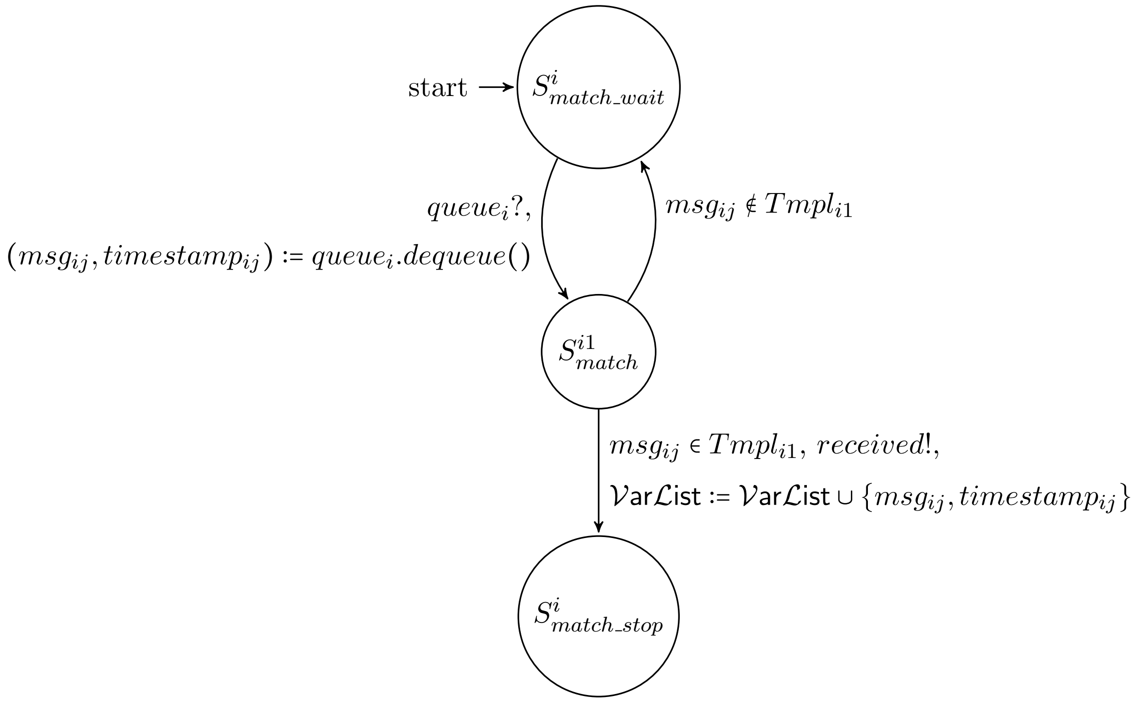

start
$S^i_{match\_wait}$
$queue_i?$,
$(msg_{ij}, timestamp_{ij}) := queue_i.dequeue()$
$msg_{ij} \notin Tmpl_{i1}$
S^{i1}_{match}
$msg_{ij} \in Tmpl_{i1}$, $received!$,
$\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} := \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} \cup \{msg_{ij}, timestamp_{ij}\}$
$S^i_{match\_stop}$

FIGURE 4.10: Match Automaton.

on this channel, for this `receive` instruction. The time stamp associated with the arrival of msg_{ij} is saved in variable $timestamp_{ij}$, which takes its value from the variable v_{now} after sending a time refresh request to the `now` automata which is active in the background. Signal $queue_i!$ is being sent to the `match` automata for indicating that a new message is available for the check. The relationship between `receive` and `match` can also be described in terms of the classical producer - consumer situation, where the `receive` represents the producer and `match` represents the consumer.

We consider that each channel (ch_i) has a queue ($queue_i$) associated with it, where the incoming messages for that channel will be stored. When `match` automata is woken up by the $queue_i!$ signal it extracts the newest message from the $queue_i$ queue and starts the comparison against the given template. We are going to further see in this chapter that there can be more than one template for messages associated with each queue. But in this case, for the purpose of this instruction, there is only one template to be matched. If the matching succeeds, both the value of the message and the time stamp for the message are saved into the $\mathcal{V}$ar$\mathcal{L}$ist, and the $receive!$ signal is then sent from the `match` automata to the `receive` automata to indicate that the right message was received and that it can move forward to the next state (see Figure 4.9). As we know, the `receive` statement is a blocking operation which returns only when the expected message is received.

If the message could not be matched, the `match` automata goes back to the waiting state until the next awakening signal (see Figure 4.10).

In the presented timed automata, $S_{receive}$ and S_{next} are symbolic states designating any generic `receive` statement, and all the possible states that come next to it. Also $S^i_{rcv\_timestamp}$, $S^i_{start\_match}$, $S^i_{match\_wait}$ and S_{match_stop} are generic states which are indexed after the number of the port they are associated with. S^{i1}_{match} state is indexed after the number of the port and the number of the template for incoming communication on the port that the state is associated with.

One possible trace for `receive` state automata might look like
$(ch_i(e?)now!queue_i!)^k received?$,
while one possible trace for `match` should be a complementary trace of the form
$(queue_i?)^k received!$;
where $k \in \mathbb{N}$ represents the number of messages received until the last one is matched.

4.5.3 Semantics Of `send` With `timestamp`

The semantics of `send` with `timestamp` is simpler than that of the `receive` with `timestamp`. It can be represented by a single timed automaton, as in Figure 4.11. The logic starts from the initial state S_{send}. After the message is sent out through the indicated port, the global time is requested using the signal $now!$, and the returned value is saved in the list of global variables. S_{send} is a generic state, indicating the initial state of any `send` operations. $S^{i}_{snd\_timestamp}$ is a generic state indicating an intermediary step for the timestamping procedure associated with a specific port; the association between the port and the state is realized through the index i. S_{next} represents a generic state, indicating the next flow of instructions.

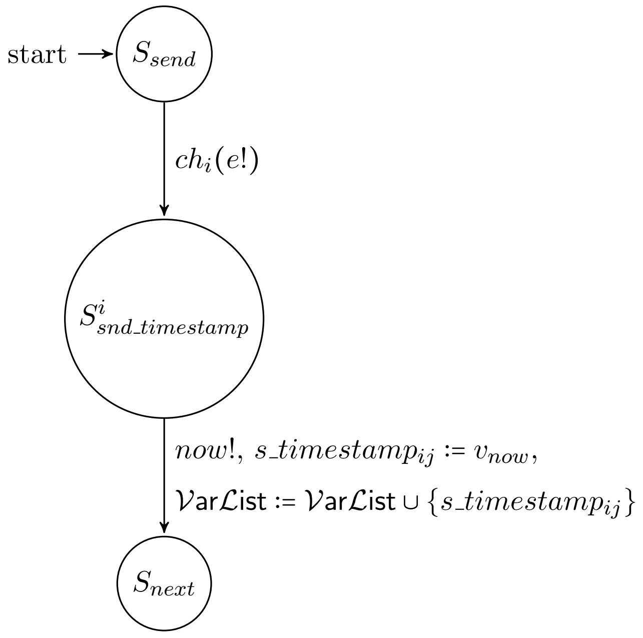

start
S_{send}
$ch_i(e!)$
$S^{i}_{snd\_timestamp}$
$now!,\ s\_timestamp_{ij} := v_{now},$
$\mathcal{V}ar\mathcal{L}ist := \mathcal{V}ar\mathcal{L}ist \cup \{s\_timestamp_{ij}\}$
S_{next}

FIGURE 4.11: Send with timestamp automaton.

A trace for the `send` automata would simple look like: $ch_i(e!)now!$.

4.5.4 Semantics For The `receive` Instructions Which Verify Incoming Communication

The semantics of the `receive` instructions which verify incoming communication using time predicates is going to be expressed by enhancing the `receive` and `match` timed automata that were introduced in Section 4.5.2.

The intention behind using time predicates in combination with the receive statement is to impose time restrictions for the arrival of messages. For verifying a real-time system it is not sufficient to verify the functionality aspects, reflected in a black-box test system by the accuracy of the responses of the *SUT* to certain stimuli, but also the timing of the responses from the *SUT*. This implies that there are two matchings performed when a new message is received: message matching against a structural template and a temporal matching. The temporal matching verifies whether or not the time predicate associated with the `receive` instruction is satisfied by the time the message is received.

The `receive` timed automaton presented in Figure 4.12 is similar to the `receive` timed automaton presented in Figure 4.9, excepting the S_{error} which is newly introduced. This is added in order to avoid the situation when a `receive` operation blocks for an indefinite period of time. If the expected message never arrives, this blocking behavior might compromise the correct functioning of the whole test system. Introducing time restrictions for the incoming messages helps avoid this situation. If the time interval when a valid message is expected is overstepped, then it becomes clear that the time predicate can not be further satisfied. In this case, the `receive` returns from the waiting state and enters an error state, where the failure of the SUT can be acknowledged.

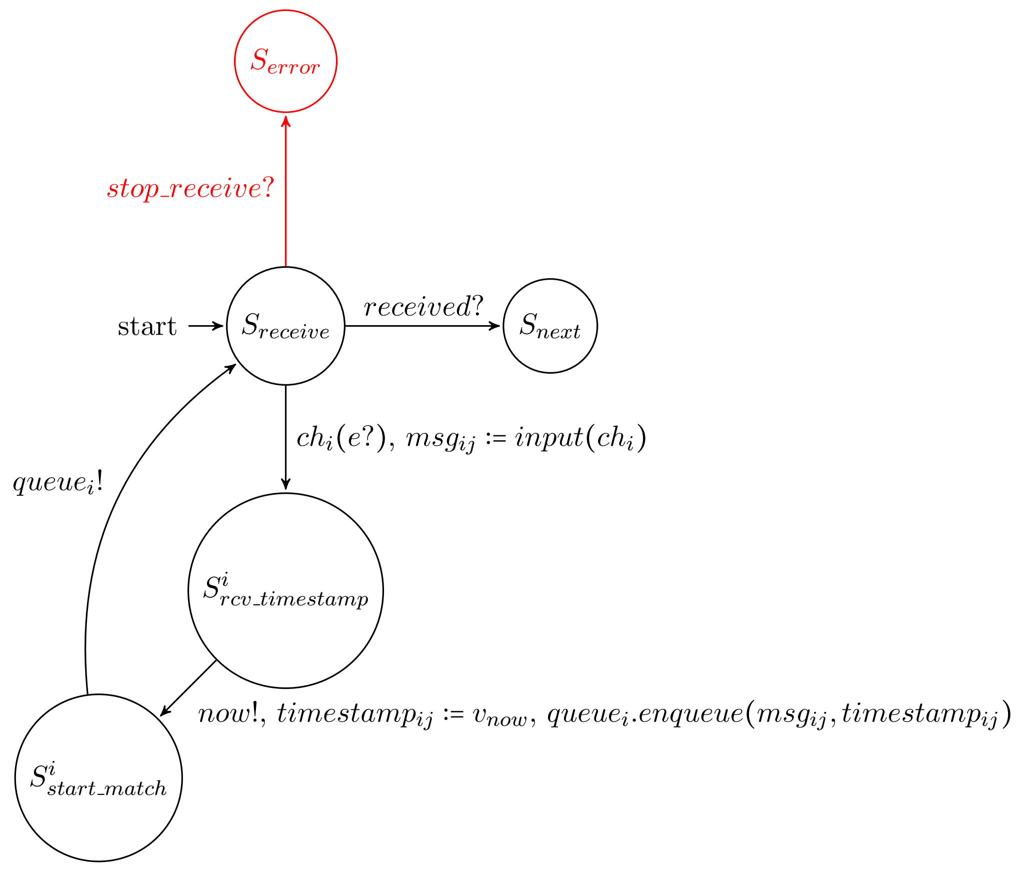


FIGURE 4.12: Receive automaton.

The two new `receive` and `match` timed automata are also complementary to each other, in the same way that it was shown for their predecessors 4.9, 4.10.

The `match` timed automata from Figure 4.10 is also enhanced with an additional state, S_{match_time}, which performs the second matching, the time matching. As illustrated here, time matching is performed before the structural matching. If the time does not correspond to the time predicate then there is no reason for continuing with the structural matching. If the time predicate is still valid (respects the validity criterions – see D.18, D.20, D.22, D.24, D.26, D.27) then the `match` automata returns to the waiting state, $S^i_{match\_wait}$. Otherwise, the `match` automata goes to an error state, S_{error},

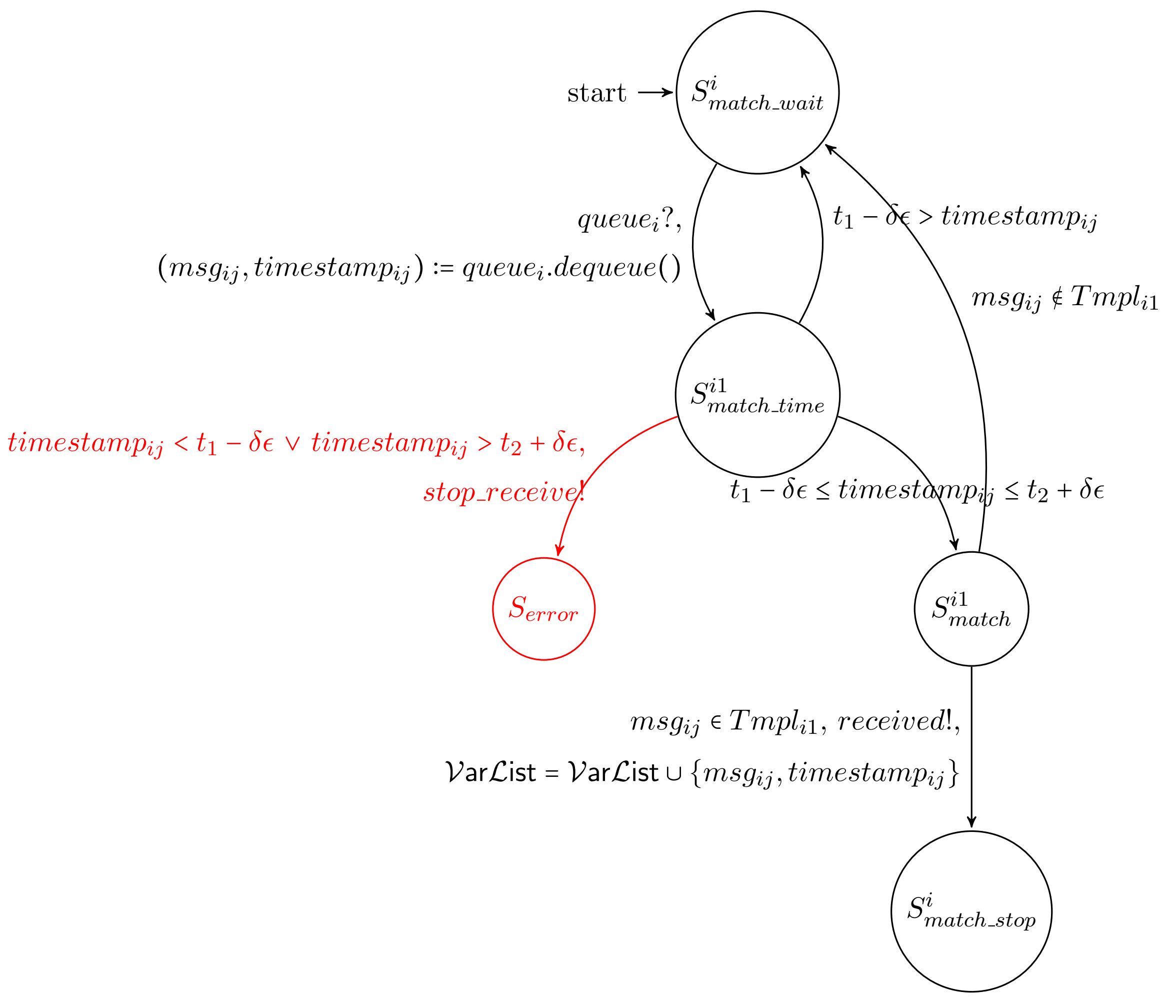

start
$S^{i}_{match\_wait}$
$queue_i?$,
$(msg_{ij}, timestamp_{ij}) := queue_i.dequeue()$
$t_1 - \delta\epsilon > timestamp_{ij}$
$msg_{ij} \notin Tmpl_{i1}$
$S^{i1}_{match\_time}$
$timestamp_{ij} < t_1 - \delta\epsilon \vee timestamp_{ij} > t_2 + \delta\epsilon$,
$stop\_receive!$
$t_1 - \delta\epsilon \leq timestamp_{ij} \leq t_2 + \delta\epsilon$
S_{error}
S^{i1}_{match}
$msg_{ij} \in Tmpl_{i1}$, $received!$,
$\mathcal{V}ar\mathcal{L}ist = \mathcal{V}ar\mathcal{L}ist \cup \{msg_{ij}, timestamp_{ij}\}$
$S^{i}_{match\_stop}$

FIGURE 4.13: Match automaton.

not before sending a signal to the associated `receive` automata to indicate that it should cease waiting for messages and go to an error state as well. S_{error} is a generic state indicating error. It is used both with the `receive` and `match` automata and is a terminal state. This means that the execution of the automata ends here.

The time predicate used for the simplicity of illustration is an `within` predicate of the form:$t_1 - \delta\epsilon \leq timestamp_{ij} \leq t_2 + \delta\epsilon$, where t_1 and t_2 are the given parameters of the constraint and $\delta\epsilon$ defines the allowed inaccuracy.

One possible trace for the `receive` automata presented in Figure 4.12 is:
$(ch_i(e?)now!queue_i!)^{k_1}received?$
and the complementary trace based on the `match` automata would be:
$(queue_i?)^{k_1}received!$,
where $k_1 \in \mathbb{N}^{>0}$ represents the number of messages received on the port and handled by this `receive` instruction until one of them is matched. This trace indicates a situation

when the SUT passed the verification that regarded both time and structure of the message.

Another possible couple of traces, this time corresponding to a failure of the SUT, might look, on the `receive` automata side, like:
$(ch_i(e?)now!queue_i!)^{k_2} stop\_receive?$
and on the `match` automata side:
$(queue_i?)^{k_2} stop\_receive!$,
where $k_2 \in \mathbb{N}^{\geq 0}$ represents the number of messages received on the port, and is handled by the `receive` instruction until the valid time frame for time constraints on the message receive finally expires.

4.5.5 Semantics For `send` Instructions Which Control Outgoing Communication

For the semantics of `send with time constraints` operation, the timed automaton illustrated in Figures 4.14 is used. The `send` timed automata waits in the starting state for the time point given as parameter for the `at` statement to be reached. This `send` automata has a behavior which is similar to the `wait` automata presented in Figure 4.5. The time point would be expanded to a time interval in the vicinity of the given time point, $[t_{max} - \delta\epsilon, t_{max} + \delta\epsilon]$, in order to introduce some toleration for error, which is inherent to the real world. If there are scheduling problems, due to overloading of the system, or other causes, and the time interval for sending the message is missed, the `send` automata enters a terminal error state. This state indicates that the $\mathcal{TS}$ itself has had a malfunction.

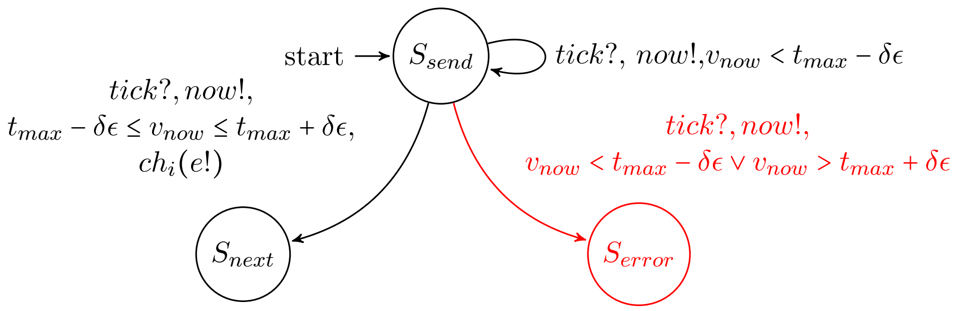

start
S_{send}
$tick?, now!, v_{now} < t_{max} - \delta\epsilon$
$tick?, now!,$
$t_{max} - \delta\epsilon \leq v_{now} \leq t_{max} + \delta\epsilon,$
$ch_i(e!)$
$tick?, now!,$
$v_{now} < t_{max} - \delta\epsilon \vee v_{now} > t_{max} + \delta\epsilon$
S_{next}
S_{error}

FIGURE 4.14: Timer For `send` State With Time Constraint.

One possible trace for the `send with time constraints` timed automata would look like:
$(tick?now!)^k ch_i(e)!$, where $k \in \mathbb{N}$ is a natural number indicating the number of clock ticks that passed before sending the message.

Another possible trace, this time indicating an error would be: $tick?now!$ which indicates that the indicated moment of time has already past.

4.5.6 Semantics for `alt` Instructions which Control Incoming Communication

The semantics of an `alt` statement is more complex than the semantics of the other instructions extended in this thesis. It involves the collaboration of the three timed automata presented in this section in Figures 4.15, 4.16 and 4.17.

Due to its complexity, Figure 4.15 shows only an excerpt from the `alt` automata. The excerpt contains the semantics associated with one representative branch of a generic `alt` statement. We assume that the generic `alt` statement considered here has a_n number of receiving branches. Each receive branch is waiting for input on one specific port of the *TS*.

We consider, as defined in Section 4.5.1, to have a function $\mathcal{B} : \mathcal{Q}\mathsf{ueues} \times \mathcal{G}\mathsf{uards} \times \mathcal{T}\mathsf{emplates} \rightarrow \mathcal{B}\mathsf{ranches}_{\mathcal{A}\mathsf{lt}}$ that represents a bijective relation, associating one queue with time guards and templates to each branch of the alternative. $\mathcal{B}(queue_i, TP_{il}, Tmpl_{il}) = branch_j$, where $i = 1..n_{ports}, l = 1..n_{tmpl_i}, j = 1..|\mathcal{B}\mathsf{ranches}_{\mathcal{A}\mathsf{lt}}| = a_n$.

In Figure 4.15, representing the main functionality of an alternative, we can see that the automata might receive input messages on different channels or ports. The flow corresponding to a receive on a port is similar for the `receive` automata, already presented in Section 4.5.4, Figure 4.12. If one message arrives while inside an alternative, the transition associated with the channel on which the message is received is going to be taken. This will lead to a state that will take the time stamp for the message arrival, and the associated `match` automata for that channel will be woken up with the corresponding signal, $queue_i$. The automata then enters back the listening state. S_{alt} state might be regarded as the state where the automata is listening to all the ports on which it expects to receive messages inside the alternative.

One possible trace for the `alt` automata will be:
$((ch_1?now!queue_1!)^*(ch_2?now!queue_2!)^*...(ch_i?now!queue_i!)^*...(ch_{a_n}?now!queue_{a_n}!)^*)^*received_{il}?$

with the complementary trace for the extended `match` automata:
$(queue_1?^*queue_2?^*...queue_i?^*...queue_{a_n}?^*)^*received_{il}!$

and the complementary trace for the `wait_alt` automata:
$(tick?now!)^*$.

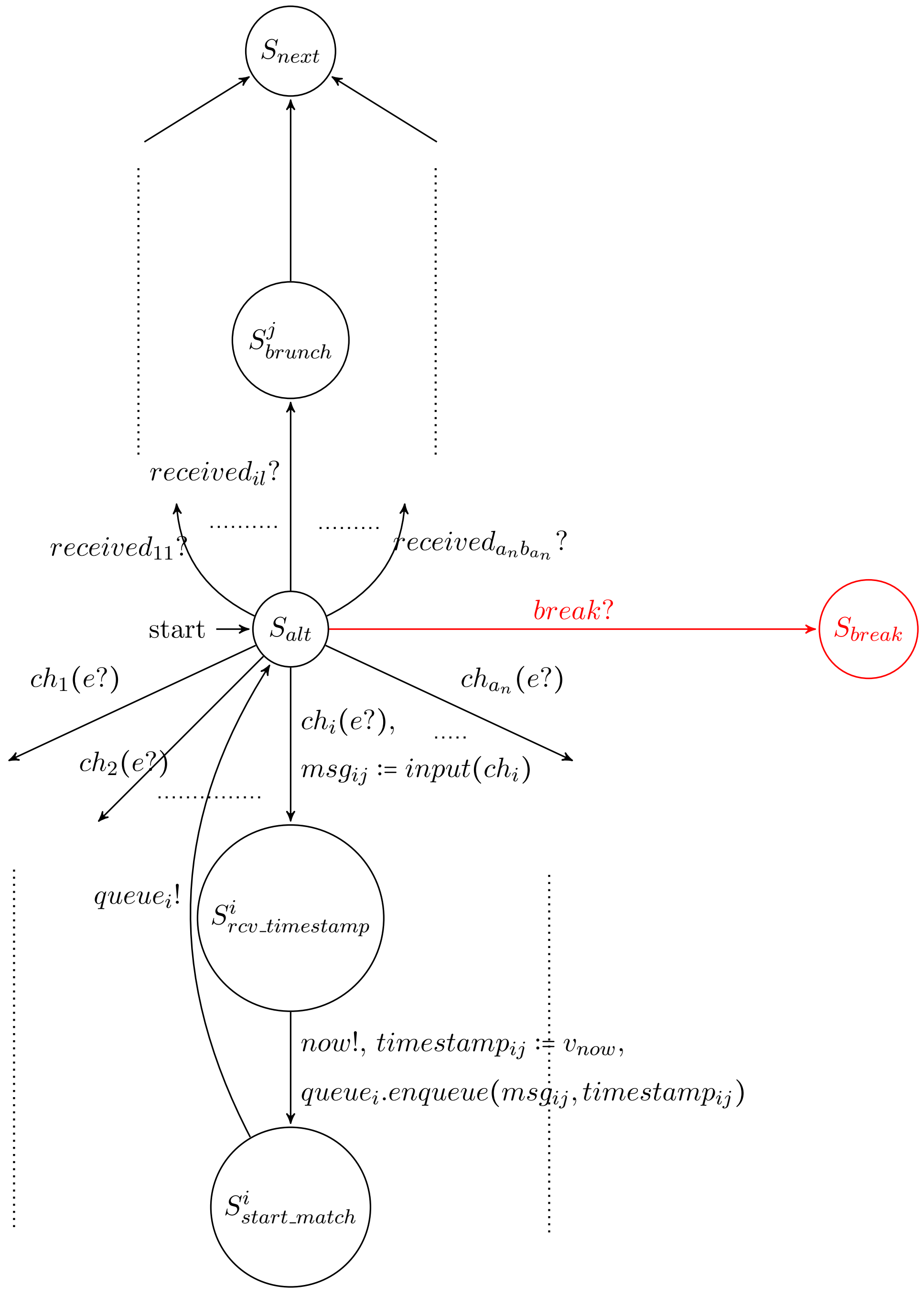


FIGURE 4.15: `alt` Automaton.

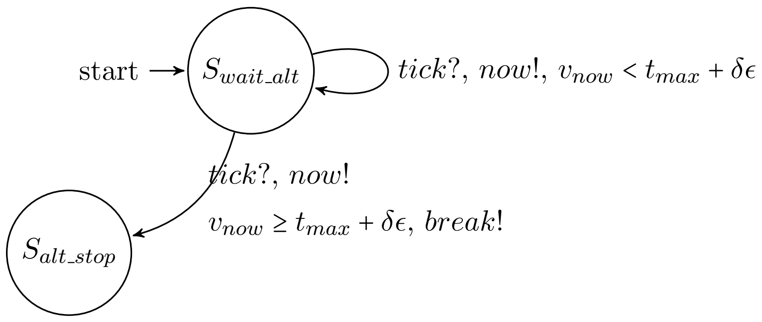

start
S_{wait_alt}
$tick?,\ now!,\ v_{now} < t_{max} + \delta\epsilon$
$tick?,\ now!$
$v_{now} \geq t_{max} + \delta\epsilon,\ break!$
S_{alt_stop}

FIGURE 4.16: Timer For The `alt` State.

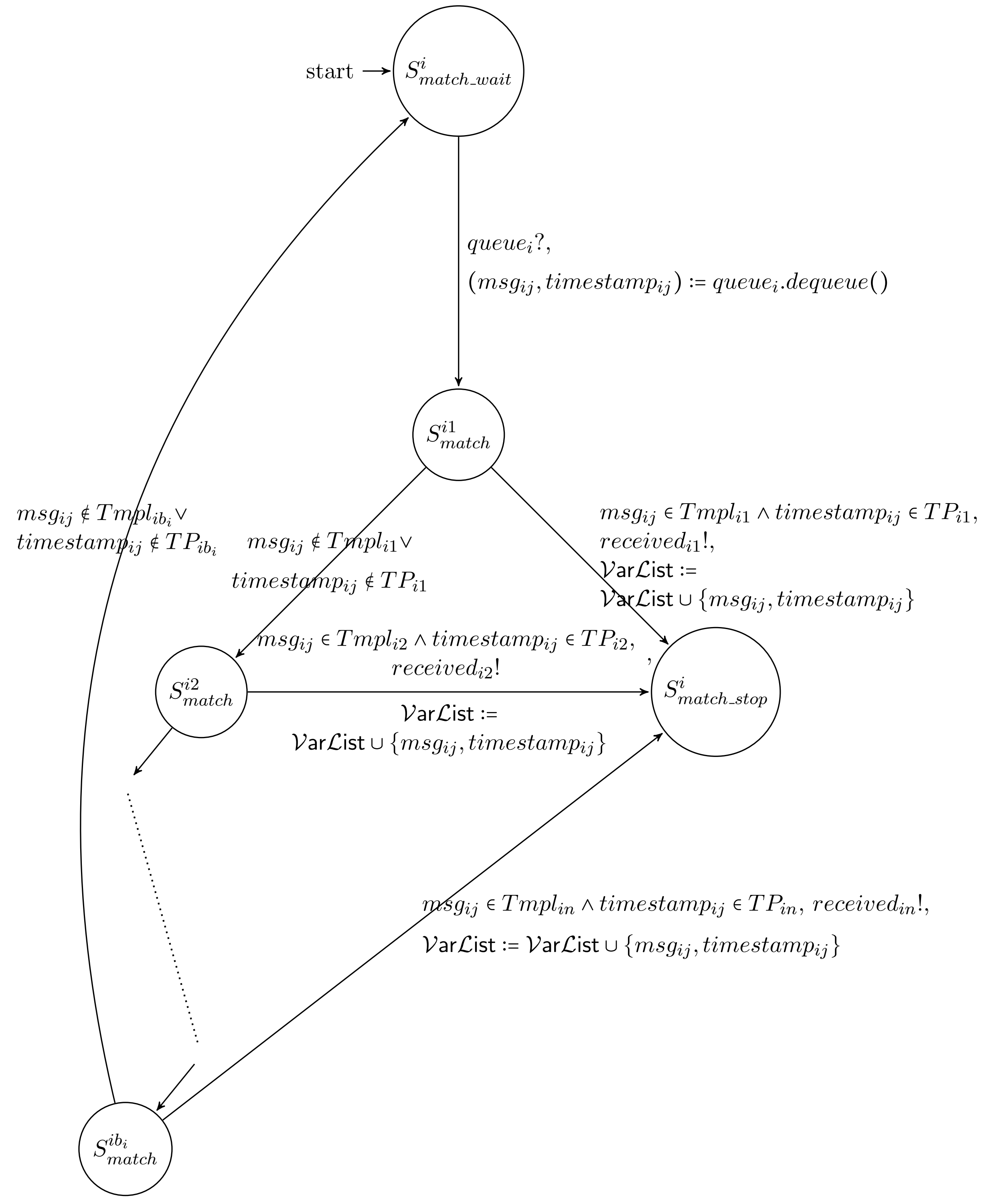

start
$S^{i}_{match\_wait}$
$queue_i?$,
$(msg_{ij}, timestamp_{ij}) := queue_i.dequeue()$
S^{i1}_{match}
$msg_{ij} \notin Tmpl_{ib_i} \vee$
$timestamp_{ij} \notin TP_{ib_i}$
$msg_{ij} \notin Tmpl_{i1} \vee$
$timestamp_{ij} \notin TP_{i1}$
$msg_{ij} \in Tmpl_{i1} \wedge timestamp_{ij} \in TP_{i1}$,
$received_{i1}!$,
$\mathcal{V}ar\mathcal{L}ist :=$
$\mathcal{V}ar\mathcal{L}ist \cup \{msg_{ij}, timestamp_{ij}\}$
$msg_{ij} \in Tmpl_{i2} \wedge timestamp_{ij} \in TP_{i2}$,
$received_{i2}!$
,
S^{i2}_{match}
$S^{i}_{match\_stop}$
$\mathcal{V}ar\mathcal{L}ist :=$
$\mathcal{V}ar\mathcal{L}ist \cup \{msg_{ij}, timestamp_{ij}\}$
$msg_{ij} \in Tmpl_{in} \wedge timestamp_{ij} \in TP_{in}$, $received_{in}!$,
$\mathcal{V}ar\mathcal{L}ist := \mathcal{V}ar\mathcal{L}ist \cup \{msg_{ij}, timestamp_{ij}\}$
$S^{ib_i}_{match}$

FIGURE 4.17: Extended Match Automaton.

One other possible trace for the `alt` automata – this time indicating failure – will be:
$((ch_1?now!queue_1!)^*(ch_2?now!queue_2!)^*...(ch_i?now!queue_i!)^*...(ch_{a_n}?now!queue_{a_n}!)^*)^*break?$

with the complementary trace for the extended `match` automata:
$(queue_1?^*queue_2?^*...queue_i?^*...queue_{a_n}?^*)^*$

and the complementary trace for the `wait_alt` automata:
$(tick?now!)^*break!$.

4.5.7 Semantics For Instructions Controlling The Starting And Stoping Of Test Components

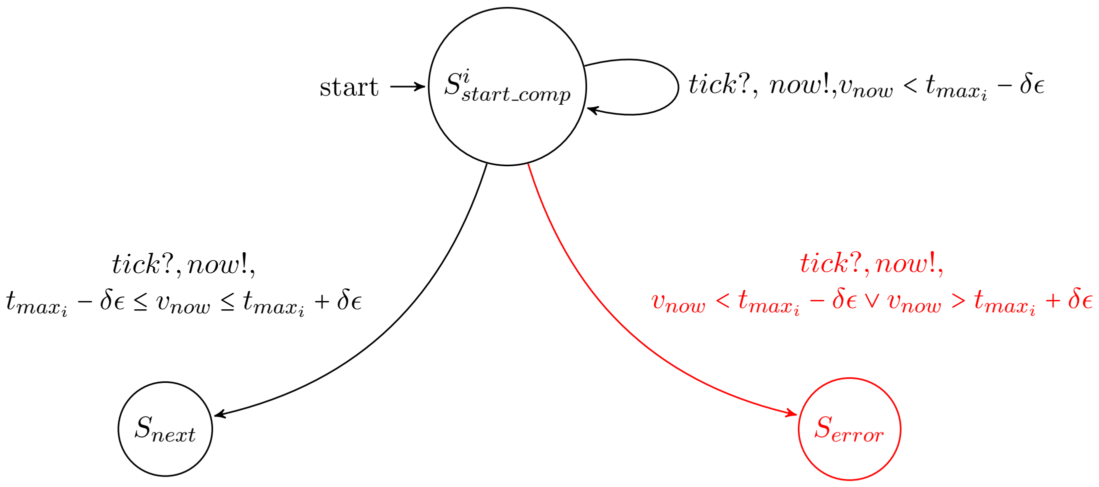

start
$S^i_{start\_comp}$
$tick?,\ now!, v_{now} < t_{max_i} - \delta\epsilon$
$tick?, now!,$
$t_{max_i} - \delta\epsilon \le v_{now} \le t_{max_i} + \delta\epsilon$
$tick?, now!,$
$v_{now} < t_{max_i} - \delta\epsilon \lor v_{now} > t_{max_i} + \delta\epsilon$
S_{next}
S_{error}

FIGURE 4.18: Timer For `start` Component State With Time Constraint.

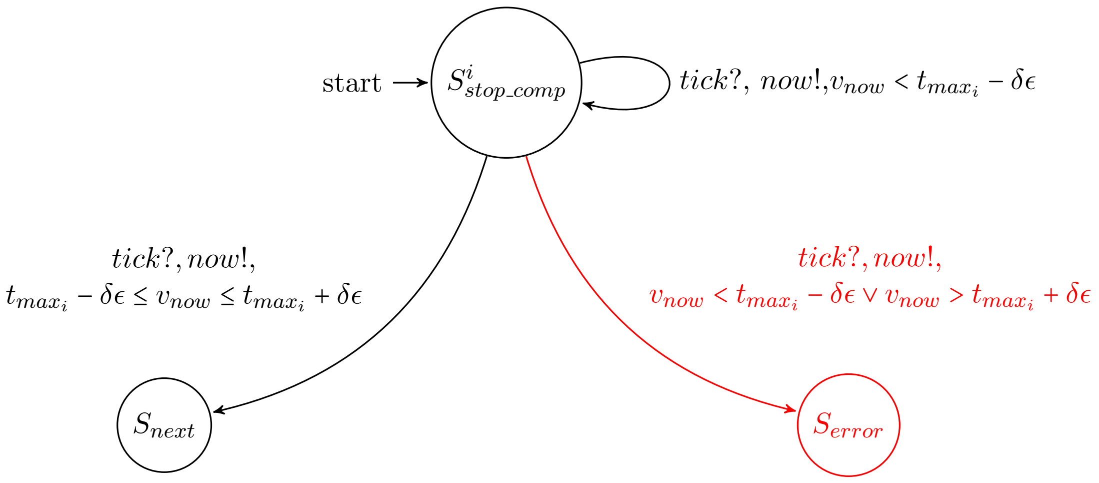

start
$S^i_{stop\_comp}$
$tick?,\ now!, v_{now} < t_{max_i} - \delta\epsilon$
$tick?, now!,$
$t_{max_i} - \delta\epsilon \le v_{now} \le t_{max_i} + \delta\epsilon$
$tick?, now!,$
$v_{now} < t_{max_i} - \delta\epsilon \lor v_{now} > t_{max_i} + \delta\epsilon$
S_{next}
S_{error}

FIGURE 4.19: Timer For `stop` Component State With Time Constraint.

4.6 Comparison Between Real-time Extended TTCN-3 And TTCN-3

In this section, we are going to demonstrate how the previously discussed real-time extensions increase the semantic power of TTCN-3 with respect to real-time situations, and why are they necessary. Therefore, we will first try to describe several real-time situations, using TTCN-3 language as it is right now. We are going to prove that the TTCN-3 specification will produce inaccurate time calculation and therefore, is an insufficient means of describing real-time requirements. We are then specifying the same examples using the extended version of TTCN-3 and thus, through comparison, reveal the benefits.

The main issues presented in this section rely on the `alt` statement with timing restrictions for the receiving of messages. The main behavior structure we are refereing to is presented by means of a flow graph representation in Figures 4.20, 4.21 and 4.22. This representation is similar to the one used to describe the operational semantics of TTCN-3 [5], and the elements of the graph are used as described within the standard. Figure 4.20 shows the overall behavior of the `alt` statement. For the purpose of keeping the conciseness, we chose to simplify the semantic of the `alt` statement and to concentrate only on the `receive` and `timeout` branches which are detailed further.

The segment from Figure 4.21 presents the main operations that are performed on the lower levels when evaluating a `receive` branch. These operations are performed over a data model consisting of the list of ports for communication, each port possessing a queue of incoming messages, and a list of timers. The usage of timers for indicating time expiration conforms to the TTCN-3 standard language. Considering this structure, there may be more than one `receive` branch and more than one `timer` to be evaluated for timeout. On each occasion when evaluating the branches of the `alt` statement, a previously taken snapshot of the data configuration of the system is being used. This means that the values from lists and queues, as well as the absolute time of the system are frozen, and remain unchanged and available for manipulation until the next snapshot. The snapshots are being taken at regular intervals of time. When entering the `receive` branch, the queue of the associated receiving port is being verified, whether it contains a message or not. If there is a new message, this is dequeued and sequentially decoded. Then, the decoded message is matched against the specified templates. If it matches, usually a statement block of instructions is expected to be executed; otherwise, if neither template matches the message, the *TS* continues its behaviour, either by waiting for a new message to come or by indicating a failure.

Alternatively, the `timeout` branch is considered a verification mechanism, supposed to certify that a message is received in adequate time. When entering the `timeout` branch, Figure 4.22, the list of timers is checked and the time structures are searched for values.

All the values of timers are the ones saved at the moment of taking the snapshot. If the `timeout` occurred, then a fail behavior is entered. If there was no event, neither receival of a message, nor a timeout, during a snapshot frame, the sequence is repeated from the beginning: a new snapshot is taken; the data configuration is updated and frozen; the `receive` and `timeout` branches are verified once again.

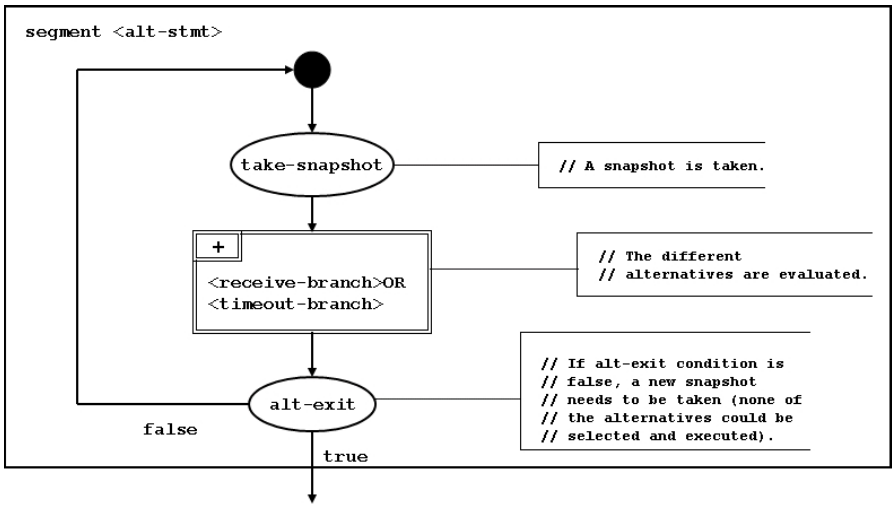


FIGURE 4.20: Flow Graph Segment For <alt-stmt> Statement [5], [4]

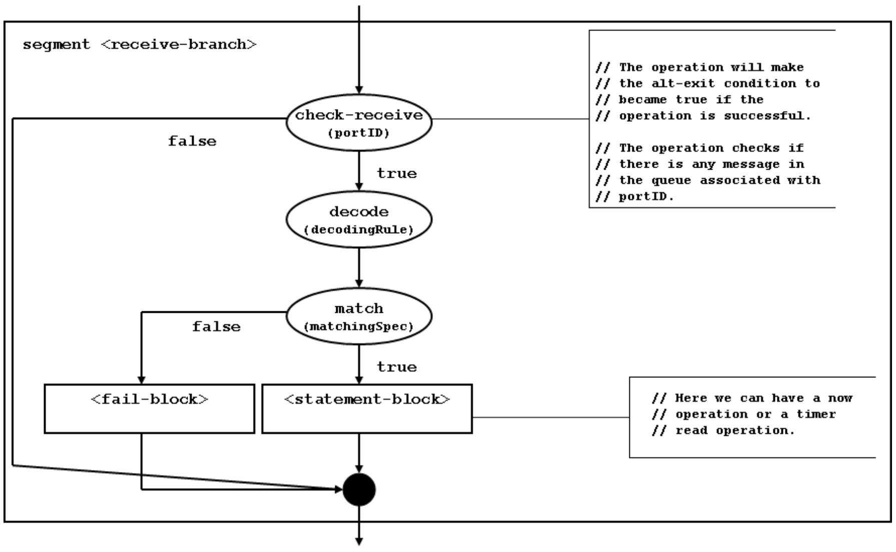


FIGURE 4.21: Flow Graph Segment For A Receiving Branch Of An <alt-stmt> Statement [5], [4]

As we will demonstrate in the following, there are some flows in the way that TTCN-3 semantics with timer have been conceived. Figure 4.23 presents sequences of events and intervals consumed by operation over the real-time axis. From a) to d), there are different possible situations presented, which are associated with verification of different time requirements. The first real-time requirement that we want fulfilled is that of

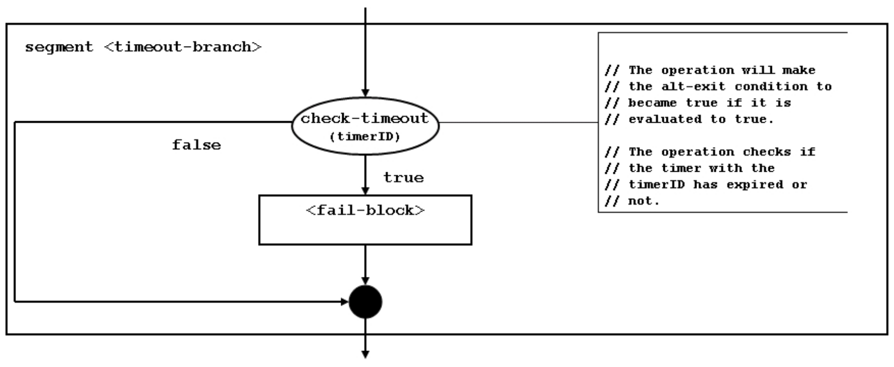


FIGURE 4.22: Flow Graph Segment For A Timeout Branch Of An <alt-stmt> Statement [5], [4]

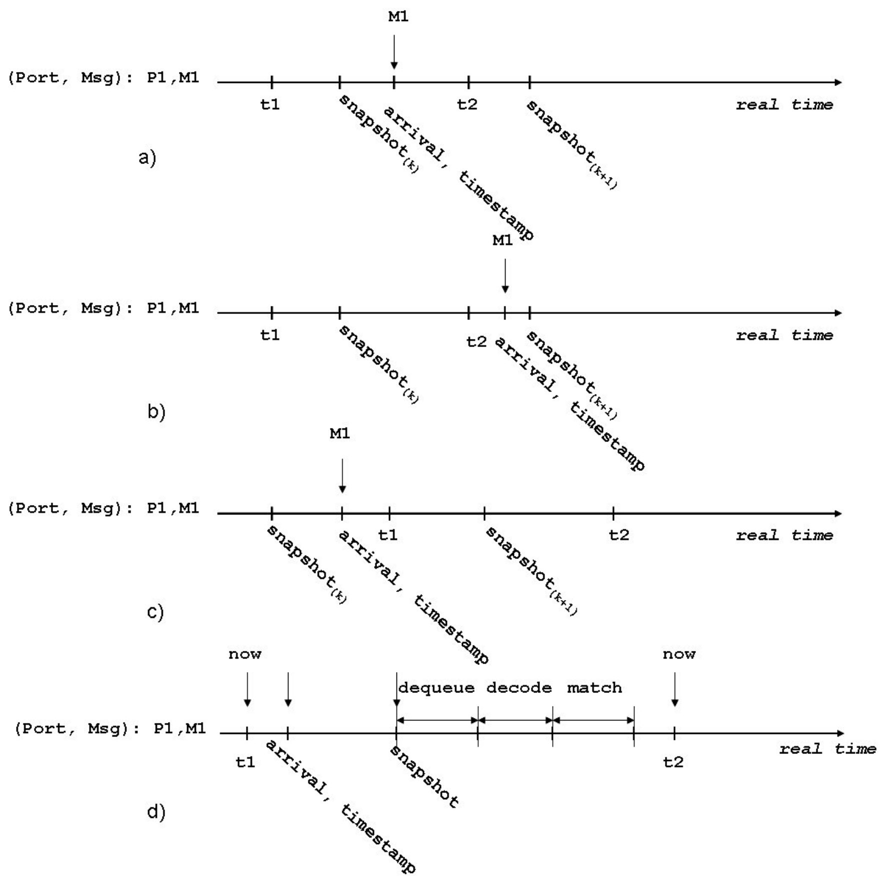


FIGURE 4.23: Events And Operations Over Time [4]

receiving a message M1 on the port P1 within strict time boundaries, for example between t1 and t2 absolute points in time.

Figure 4.23 a) and b) present two possible situations: in the a) situation, the message is received in time and in the b) situation the message is received after t2. The moment when the message is received is the moment when the message enters the queue of the system and it is automatically saved together with a timestamp. On the two charts we have also represented the moments when the snapshots are being taken, before evaluating the branch of the `alt` used to catch the message M1 on the port P1. The test solution defined with classic TTCN-3 is presented first, in Listing 4.7, and afterwards the test solution using the extended version of TTCN-3 is presented afterwards.

In the first solution we rely on the usage of a classic timer to guarantee the accomplishment of timed requirement. This timer's value is updated every time a snapshot is taken. Figure 4.23 a) shows that, in the situation in which the moment of receiving the message M1 and the upper limit of the timed requirement(t2) are both between two consecutive snapshots, we may have both a timeout and a receive event at the time of the evaluation of the second snapshot. Therefore, it depends only on the priority of the first evaluated branch that the message it is considered whether or not was received in time. If the `timeout` branch is the first one to be evaluated, it is wrongly considered that the message was not received in the proper time. In the b) situation, the message M1 is not received in time. At the same snapshot we may have again both a `receive` event and a `timeout`. Depending again on which branch is evaluated, a different verdict is established.

The second part of Listing 4.7 provides a solution which does not rely on the concept of `timer` but on the usage `within` predicate. In this situation, when evaluating the `receive` branch, together with the matching mechanism against the structural template, an additional matching is performed in order to verify if the message was received in time. The time-stamp saved at the entrance of the message into the system is compared to the bounding time values that parameterize the `within` construct. This time, the exact time value is being compared, the same one saved when the message entered the system and not the one saved when the snapshot was taken. Therefore, the evaluation is accurate.

In the situation presented in Figure 4.23 c) the message M1 is received before the lower time boundary. This situation cannot be detected using the standard TTCN-3, as there are no means for describing that. In the extended TTCN-3, this situation is covered by using the `before` construct, as illustrated in the second part of the Listing 4.7.

```
// TTCN-3 solution
timer T;
T.start(t_max);
alt{
    []P1.receive(M1){
        t_crt:=T.read();
    };
    []T.timeout(){setverdict(fail)};
}

// Extended TTCN-3 solution using within statement
// and before or after alternatives
// The difference between t1 and t2 is t_max
var datetime t1, t2;
alt{
    []P1.receive(M1) within(t1..t2){...};
    []P1.receive(M1) before t1 or after t2{
        setverdict(fail)
    };
}
```

LISTING 4.7: Timer Timeout vs. `within` statement

The third requirement is to calculate as precisely as possible the execution time that will elapse between two different points in the execution of the *TS*. The first point is the one just before entering the `alt` statement, and the other one is just after receiving a conforming message. The usage of classical timers is very imprecise, when trying to achieve this. As illustrated in the Figure 4.23 d), the current value of the timer is the same one taken at the last snapshot. Nevertheless, in the actual execution, additional delays are introduced by the `dequeue`, `decode` and `match` operations. These delays will not be acknowledged if we use the classic TTCN-3 and timers as in Listing 4.8, in the first part. In order to be accurate, these delays have to be considered as well. Using the enhanced real-time semantics we can introduce two observation points just before `alt`, and after `receive`, through the usage of `now()` operators. Values returned by these will be accurate (see Listing 4.8, second part). In addition, if we want to know the exact time of receiving a message, TTCN-3 offers no proper means for that. This can be achieved in our new semantic through the usage of the `time-stamp` as it can be seen in Listing 4.8, the third part.

4.7 Summary

This chapter has presented the real-time extensions for TTCN-3 that remain at the basis of the testing framework developed in this thesis. The chapter began with discussing the faults of TTCN-3 with regard to real-time testing, and discussed the concepts that are needed to overcome them. According to these concepts, appropriate extensions for real-time TTCN-3 have been defined. The introduced extensions were: new data types suitable for expressing time values, special operators relying on time (`now`, `wait`,

```

// TTCN-3 solution
timer T;
T.start(t_max);
alt{
    []P1.receive(M1){
        t_elapsed:=t_max-T.read();
        if(t_elapsed > t_limit){...};
    };
    []T.timeout(){setverdict(fail)};
}

// Extended TTCN-3 solution with now() operation
var datetime t1, t2;
t1:=now();
alt{
    []P1.receive(M1) {
        t2:=now();
        if((t2-t1) > t_limit){...};
        ...
    };
    ...
}

//Extended TTCN-3 solution with timestamp operation
var datetime t1, t2;
t1:=now();
alt{
    []P1.receive(M1)->timestamp t2 {
        if((t2-t1) > t_limit){...};
        ...
    };
    ...
}
```

Listing 4.8: Timer read vs. `now()` operation and vs. `timestamp` statement

`testcasestart`, `testcomponentstart` and `testcomponentstop`), measurements of time with `timestamp`, time restrictions for message receival using time predicates (`at`, `within`, `before`, `after`), induction of events at established time points using the time predicate `at`. A clear semantics has been described for each of the proposed extensions, by means of TA. The semantics presented here was completed by definitions based on logic rules, that have been provided in Appendix D. The semantics for time expressions with numerical and logical operators were additionally provided in Appendix C and predefined conversion functions for the different time data types representations were attached in Appendix B. All extensions were also provided with syntactic rules, expressed in EBNF notation, the same notation used to define the syntax of TTCN-3. The syntax of the extensions has been attached in Appendix A. The chapter concluded with an informal demonstration of the capabilities of the extended TTCN-3 compared with the classic TTCN-3, by means of examples. In Appendix E code samples are provided of enhanced real-time TTCN-3 specifications.

Chapter 5

Real-time Testing Framework Architecture

"Ah, to build, to build! That is the noblest art of all the arts."
Henry Wadsworth Longfellow

A framework can be seen as embodying a complete design of an application, while a pattern can be regarded as an outline of a solution for a class of problems. In our case, the class of problems is represented by real-time test systems, which are in fact communication-oriented real-time applications, meaning an application with strict timing constraints, especially concerning the input/output operations on ports.

Figure 5.1 represents the categories of patterns associated with the concepts and the implemented frameworks. The frameworks implemented as a proof of concept are based on two concrete real-time operating systems, FreeRTOS (see [115]) and RTAI Linux (see [116]) respectively.

5.1 Design Patterns For The Real-time Concepts

Before going into the details of the frameworks' realization, this chapter is going to study the main design paradigms as well as the constituting issues for the building of a real-time *TS*, based on the concepts presented in Chapter 4.

A design pattern is a formal way of documenting successful solutions to problems. Figure 5.1 presents a set of design patterns for a *RTTS* realization. These patterns have been derived from our research work and are combine what we consider to be good real-time system design practices based on our experience of realizing successful (see Chapter 7) implementations of a *RTTS* with the real-time extensions for TTCN-3 on two real-time operating system platforms (FreeRTOS and RTAI Linux).

As shown in Figure 5.1, the package containing the design patterns, represents a generalization of the package containing two frameworks implementations. Thus, the model is extensible, so that other implementations based on the patterns contained in *RTTS Design Patterns* package can be added to the *RT Testing Framework* package.

The patterns presented here are grouped in four packages, mirroring the categorizations of the real-time concepts shown in Tables 3.4- 3.8 from Section 3.3, Chapter 3:

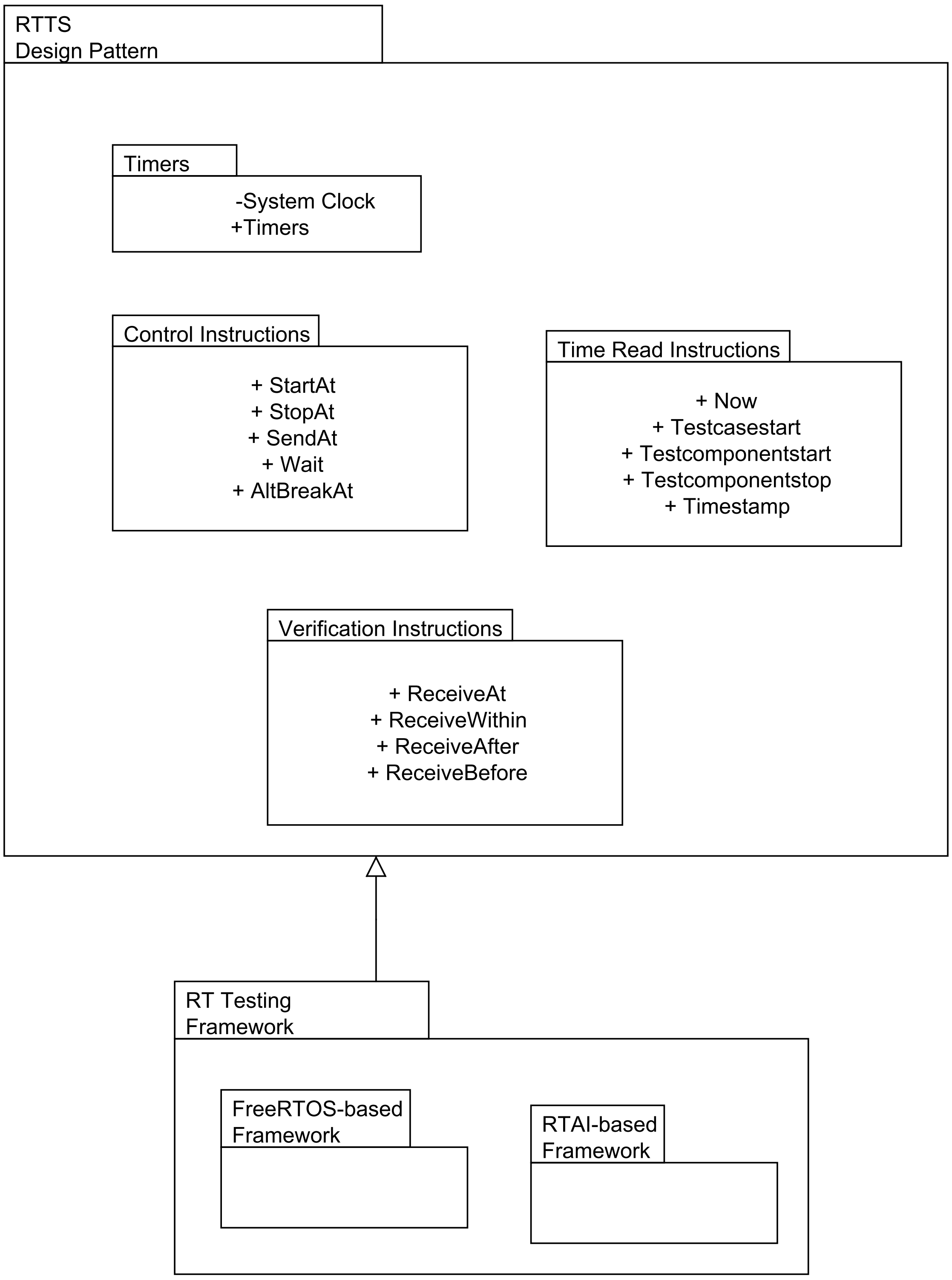


Figure 5.1: Real-time Design Patterns

1. Timers package – for *the representation of time* – contains the patterns modeling the *clock of the systems* and the associated *timers*.

In this thesis we are going to consider a solution for a non-distributed *TS*, but the problem of distribution is not going to be treated here. Therefore, the assumption of one single *clock of the system* is sufficient for time measuring.

As shown in Figure 5.1, the *clock of the system* is a private attribute. Therefore, the access to the time information counted by the *clock* is made public only through the use of *timers*.

2. Time Read Instructions package – for *the measurement of time* – contains the patterns modeling the realization of the `now`, `testcasestart`, `testcomponentstart` and `testcomponentstop` operations with the use of real-time operating system services.

3. Control Instructions package – for *control of application* – contains the patterns modeling the realization of the `start` *test component* `at`, `stop` *test component* `at`, `send at`, `wait` and `alt..break at` operations with the use of real-time operating system services.

4. Verification Instructions package – for *time verification* – contains the patterns modeling the timing mechanisms for time-stamping the incoming communication.

All these patterns will be discussed in detail in the following.

5.1.1 General Architecture

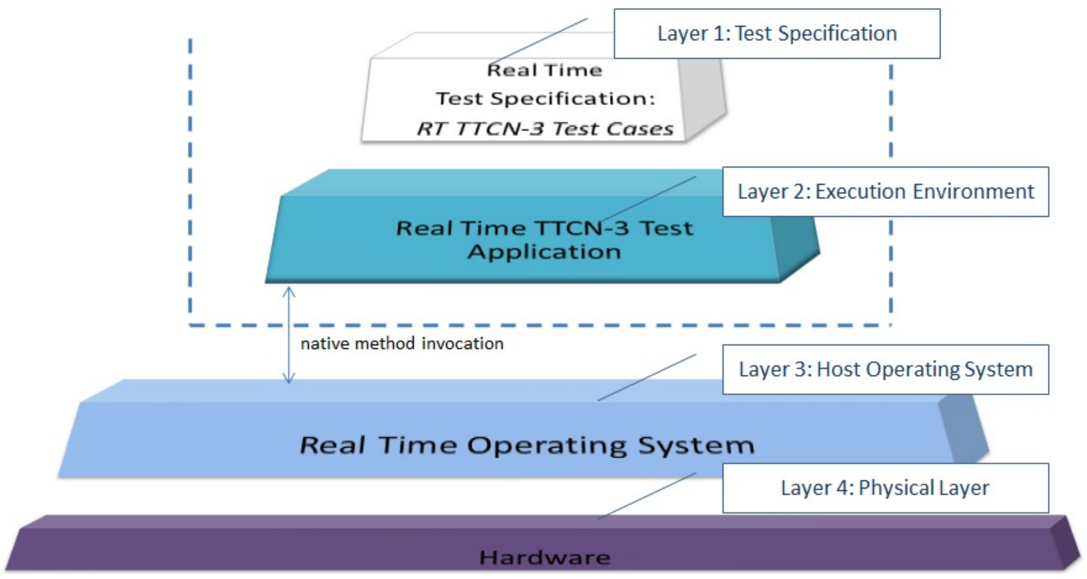

Layer 1: Test Specification
Real Time
Test Specification:
RT TTCN-3 Test Cases
Layer 2: Execution Environment
Real Time TTCN-3 Test Application
native method invocation
Layer 3: Host Operating System
Real Time Operating System
Layer 4: Physical Layer
Hardware

Figure 5.2: Architectural Layers Of A Real-time TTCN-3 Testing Framework

Figure 5.2 depicts the overall architecture of a *RTTS* designed for TTCN-3. The architecture is shown in a pyramidal form, from the highest level of abstraction (*Layer 1*) –

in our context, this is the *RTTS* designed with Real-time TTCN-3 – to the lowest level (*Layer 4*), the level of hardware and physical medium.

As presented in Section 3.2.1, Chapter 3, TTCN-3 is an abstract specification for testing and it is portable on different platforms. Using TTCN-3, the tester can focus on *test cases* definition, without worrying about the platform underneath. In order to make these *test cases* executable, they need to be converted into code that can be run as part of the *execution environment*.

The *execution environment* is depicted as *Layer 2* in Figure 5.2. The code generated for the Real-time TTCN-3 *test cases* together with its *execution environment* can be regarded as a real-time application that represents the *RTTS*. This *RTTS* runs on top of a real-time operating system (*Layer 3*) and it makes use of the services that the real-time operating system provides, allowing the management of available hardware resources (*Layer 4*).

The design patterns presented in Figure 5.1 draw the connection between *Layer 1* and *Layer 3* in Figure 5.2. Therefore, the realization of these patterns would basically be an implementation of a *RTTS* situated at *Layer 2* in Figure 5.2.

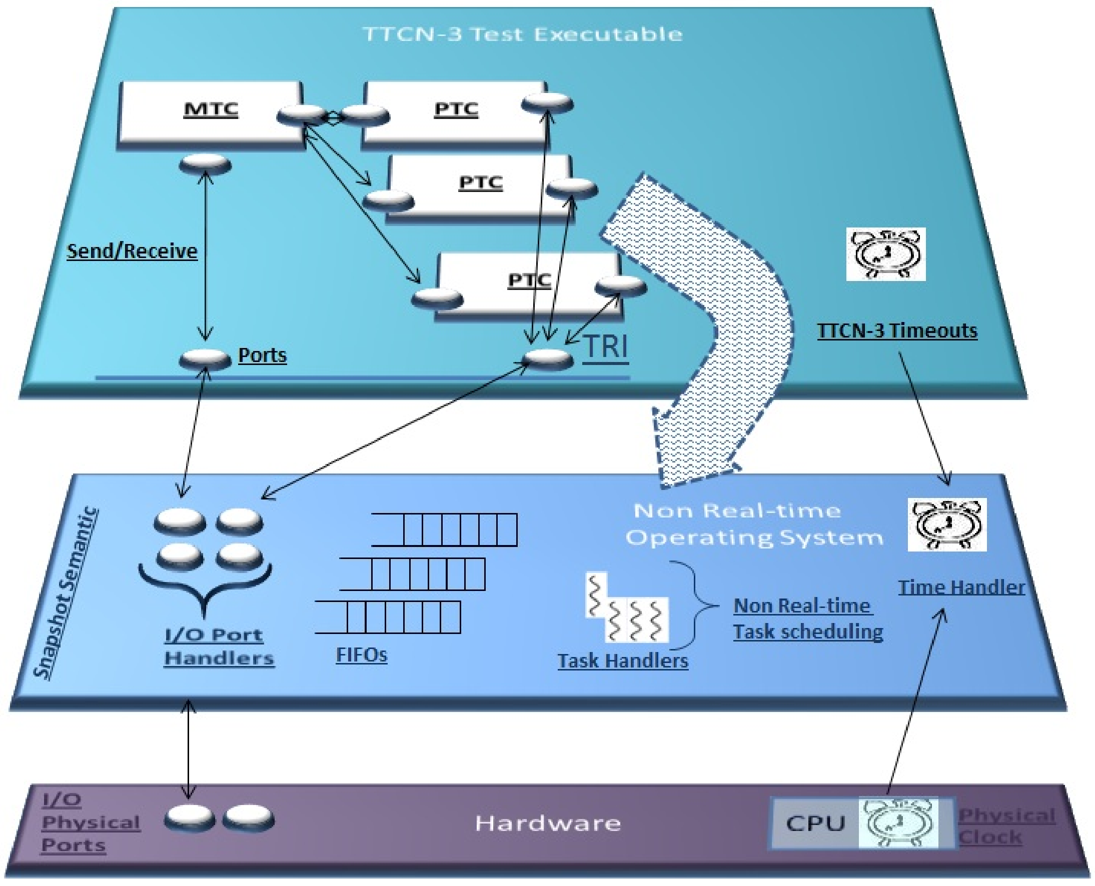


FIGURE 5.3: Implementation Design For A Conventional TTCN-3 Platform

Figure 5.3 presents the simplified runtime architecture of a classical TTCN-3 Test System (TTCN-3 *TS*). A TTCN-3 *TS* is composed at runtime from several components running in parallel, communicating with each other, as well as the environment through ports. The *MTC* is the first component created when the test case is started.

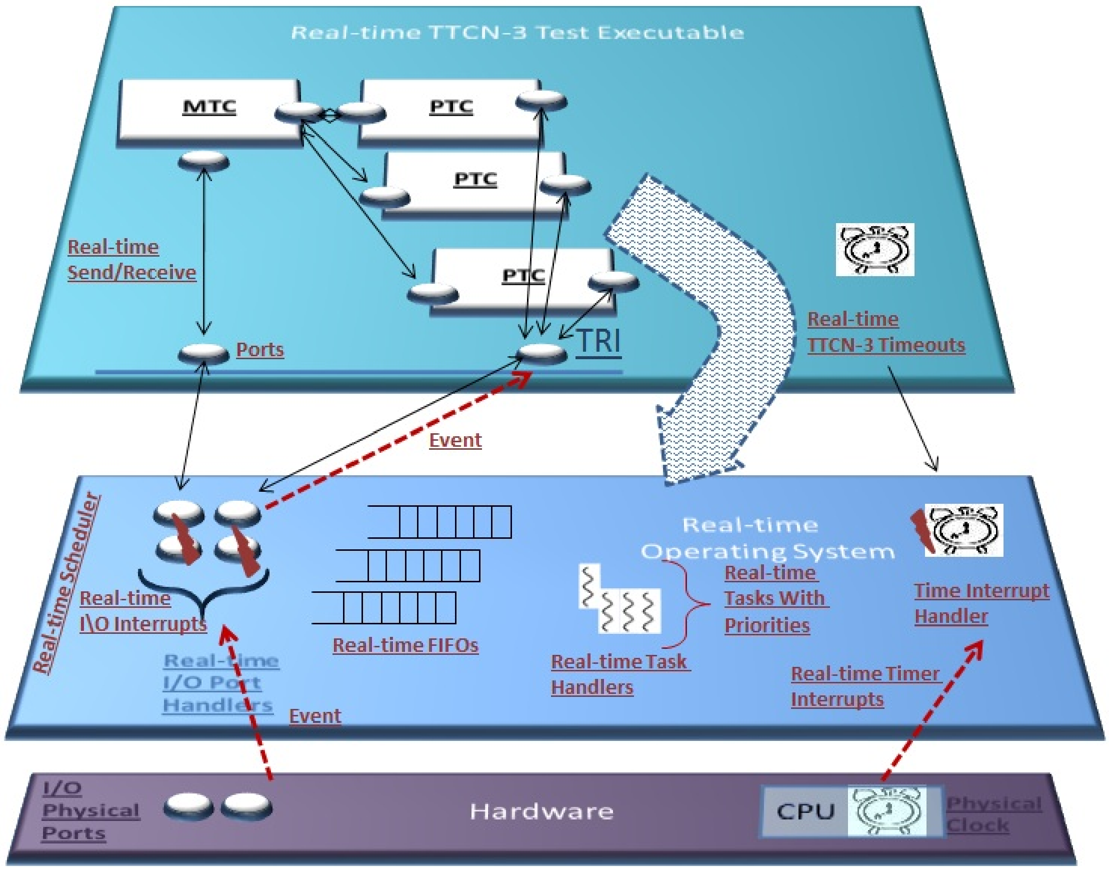


FIGURE 5.4: Implementation Design For A Real-time TTCN-3 Platform

To enhance this *TS* with real-time capabilities, we have to assure that the interactions with the SUT are handled efficiently and on time. Therefore, those interactions are enhanced by a higher priority than other parts of the *TS*'s behavior and thus, we talk about an event-triggered *TS*. In order to provide these events with timing quantifications, we need an internal *clock* which is able not only to measure the time passing, but also to generate some time events when thresholds are reached. The enhanced *TS* is illustrated in Figure 5.4.

This shows how events coming from exterior generate interrupts on ports on which they are received, and so ask for immediate handling. The inner *clock* of the *TS* must be precise and should impose a deterministic time behavior to the system. Timer interrupts are generated for assuring bounded execution times.

The *RTTS* can be regarded as a real-time application, running on a real-time operating system. Components can be seen as tasks between which we can have interdependence relations (e.g. any `PTC_Task` is created after `MTC_Task`). Therefore, operating system mechanisms such as *tasks*, *event interrupts*, *event handlers*, *clock*, *system queues*, *real-time scheduler* are going to be used for implementing the real-time TTCN-3 concepts.

5.1.2 Dealing With Time

Time points are regarded as positive real numbers, `timespan` values, `datetime` values or virtual `tick` values. Nevertheless, on a real machine they will be translated as the number of ticks of the inner clock of the system. The final values of the temporal predicates are calculated using the internal representation of time.

From a time perspective, the `send`, `receive` and `break` operations together with the time predicates and the time measurement mechanisms, presented in Chapter 4, can be divided into three categories:

- Control instructions:

  e.g. `p.send(msg) at tx;`
  `alt{...}break at ty{...}`

- Time read instructions:

  e.g. `p.send(msg) -> timestamp tx;`
  `now;`
  `comp.start-> timestamp tcx;`

- Verification instructions:

  e.g. `p.receive(msg) at tx/before tx/after tx/within(tx,ty);`

Their mechanisms are presented in detail in the following sections.

5.1.3 Control Instructions

The control instructions are the instructions that have to be performed at precise points in time. When these points in time are encountered, the clock system generates a timer interrupt. The timer interrupt is treated by an interrupt handler, which interrupts any running activity and performs the code associated with that instruction.

Such instructions are, for example, the `send` and `break` instructions with `at` time predicate. At the initialization of the test system, all the time values which are given as parameters to the `at` predicate are saved into a list maintained by the kernel (see Figure 5.5). This list is ordered by the increasing order of time values. Another step performed in the initialization phase is that of the creation of time handlers for each value in the list. The handlers have a behavior associated with the type of instruction to which the temporal predicate is applied to: `send` or `break`. When the clock tick value equals one value from the list, a time interrupt is generated and the associated event handler is invoked. Because the interrupts are not generated periodically it is said that the timer runs in one shot mode.

Component `start` and component `stop` at fixed times are also belonging to this group. These operation will be associated with corresponding handling routines.

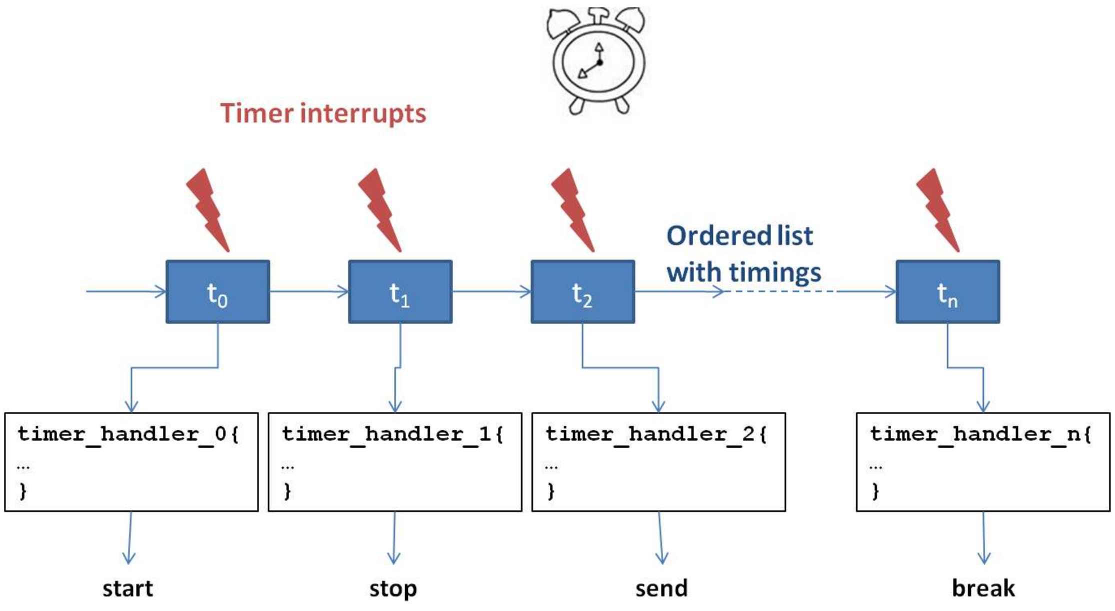


FIGURE 5.5: Generating Timer Event [6]

5.1.4 Dealing With Events

In a real-time *TS*, we are dealing with two types of event: internal generated events, for time determinism, also known as time events, and external generated events, which are triggered by the I/O ports whenever a new message has arrived. The way in which these events are to be handled will be discussed in the following section. Concerning the discussion about event handlers, it is important to mention that they have the highest priority in the system. They can interrupt any running process. Interrupt handlers will be non-preempt-able. Nevertheless, when a test is designed, the latency introduced by the system should be taken into consideration. The latency represents the amount of time that passes between an event arrival and the starting of its handling routine. For real-time operating systems this delay should be upper bounded and very small (e.g. for RTAI it is of the order of microseconds [116]).

5.1.5 Time Events

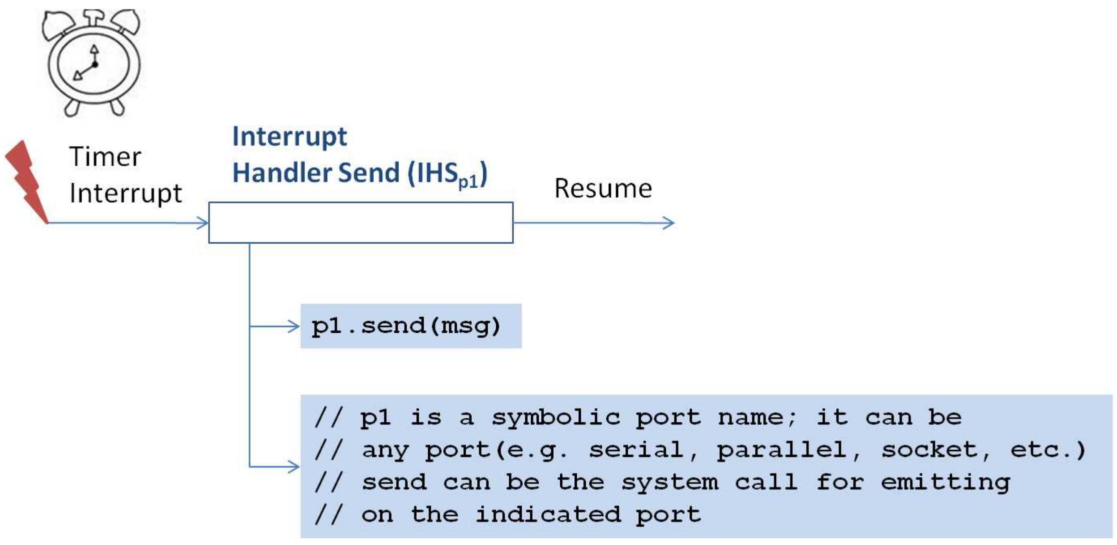

Timer Interrupt
Interrupt Handler Send (IHS$_{p1}$)
Resume
p1.send(msg)
// p1 is a symbolic port name; it can be
// any port(e.g. serial, parallel, socket, etc.)
// send can be the system call for emitting
// on the indicated port

FIGURE 5.6: Time Event Handler Associated With Send [6]

For the time events that are generated as described in 5.1.3, we have four types of event handlers: the handler associated with a `send` operation, which performs the sending of a message when it is invoked, presented in Figure 5.6, the handlers associated with forced `start` and `stop` operations of a test component – which are similar to the one for the `send` operation –, and the handler associated with the `break` instruction, which has a more complex behavior.

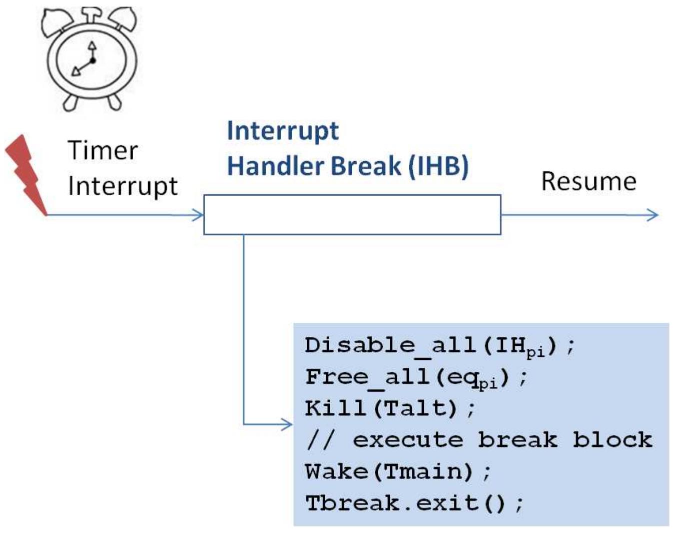
As described in Chapter 4, the `break` instruction is used to impose an upper time limit to the `alt` statement. If the `alt` statement isn't processed in time, the `break` instruction is used to bound its execution time. Therefore, the invoked handler is used also to kill the task associated with the `alt` statement. The detailed flow of events will be presented in more detail in the following sections, but the main idea is captured in Figure 5.7.

5.1.6 External Events

When a message is received from the *SUT*, an I/O interrupt is generated on the receiving port. A virtual queue is associated with each receiving port, for the purpose of storing incoming messages. When the interrupt activates the handler, the handler takes the message, reads the system's time from the system's clock, and saves the message together with the time stamp into the queue associated with the port. The procedure is presented in Figure 5.8.

5.1.7 Verification And Time Read Instructions

Verification and time read instruction do not have an intrusive action in the behavior of the application. They do not influence the scheduling mechanism, they do not cause



```
Disable_all(IHpi);
Free_all(eqpi);
Kill(Talt);
// execute break block
Wake(Tmain);
Tbreak.exit();
```

FIGURE 5.7: Time Event Handler Associated With Break [6]

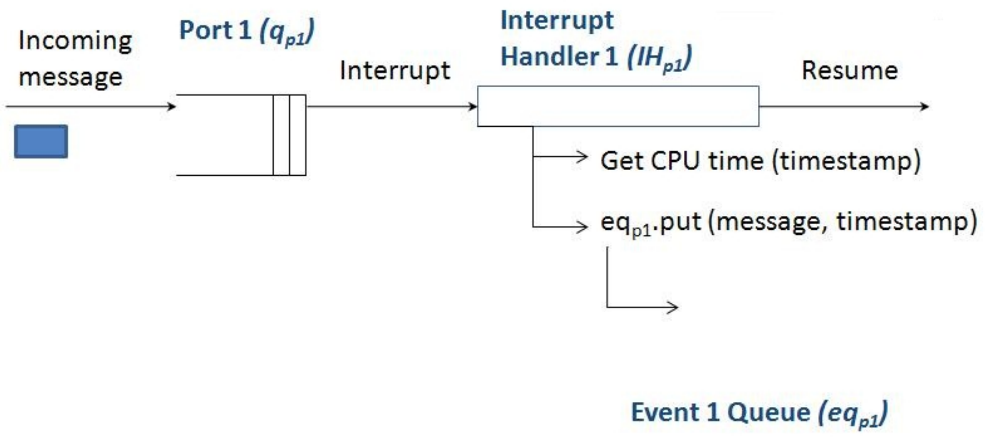


FIGURE 5.8: Event Handler For An External Event [6]

the preemption of the current running thread. Their effect is to keep a time stamp of the time when the associated operation was performed. In the case of the `receive` operation, the time stamp is taken directly from the value saved by the interrupt handler associated with the incoming event (see Figure 5.8). In the case of time read operations, the time stamp is recoded for logging and ensuring the timed trace of the test run is kept. In the case of verification operations, the time stamp is compared for conformance with a time predicate, and it will influence the verdict about the validity of the received message.

In the case of `send` operation, the behavior is a bit more complex. Only `send` with temporal predicates are implemented using interrupt handlers. `send` without temporal predicate executes just as a normal `send` operation in the context of the thread that calls it. For saving the time at which the message is send, a time primitive reads the time immediately after the message leaves the output queue of the system. The thread must not be preempted between these operations. Therefore, the interrupts have to be disabled, and maximum priority level for the task should be assured during this code section.

5.1.8 Component Task With Timed Send

Test components might be regarded as tasks, executing different pieces of system's functionality. It is interesting to analyze how a test component task is influenced by certain operations contained within the component, representing the component's behavior. The description of one test component's behavior in TTCN-3 is described in Listing 5.1.

```
testcase SendAtTime() runs on SenderComponent
{
    float  tx;
    ...
    // other computations
    senderPort.send(msg) at tx;;
    // other computations
    ...
}
```

Listing 5.1: Component task with timed send. TTCN-3 code.

Figure 5.9 presents one possible execution flow, achievable if the Send Task is the only task running in the system at that period. The `send` instruction is a control instruction and therefore, it has an impact on the state of the Send Task that is running in the system at the moment at which the `send` operation should be executed. When the time interrupt is generated, it triggers the handling routine, which preempts the Send Task and puts it in the ready queue. Ready queue is a queue maintained by the system, which contains the tasks that are ready for being scheduled next. After the routine finishes the sending operation, the scheduler gets the first available task from the ready queue and dispatches it. Usually, the ready queue is an ordered queue, sorted on different criteria such as: task priority or task deadline. Figure 5.10 shows the transition states of the Send Task.

5.1.9 Component Task With Send And Timestamp

From the execution time line presented in Figure 5.11 it can be observed that the `send` with `timestamp` operation has not an intrusive effect on the execution of the component itself. Nevertheless, after sending, the task should not be preempted by another task before keeping the time stamp. In simpler terms, it locks the processor for a short period. The associated code for the execution is listed in Listing 5.2.

TTCN-3 code associated with this behavior is listed in 5.2.

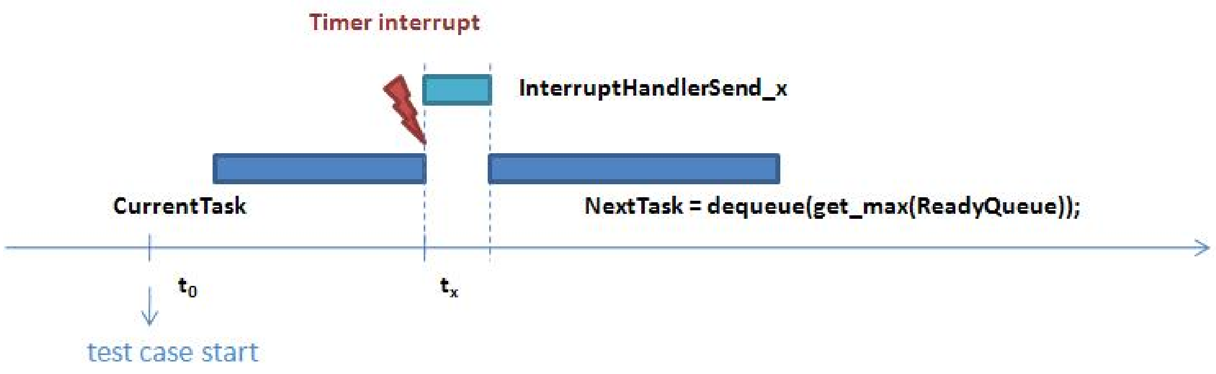


FIGURE 5.9: Execution Timeline For Sending At A Given Time [6]

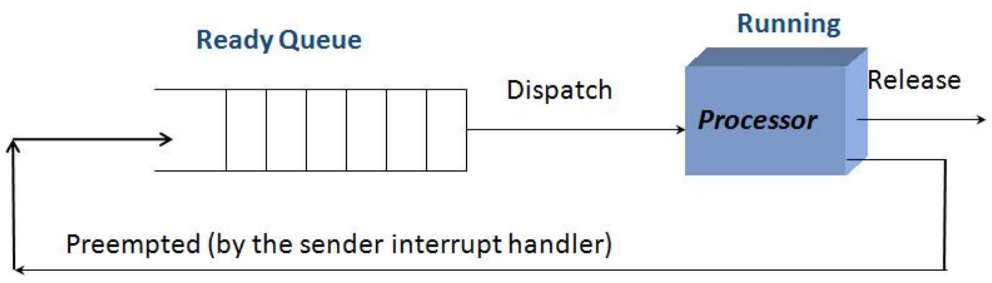


FIGURE 5.10: Sender Task Transitions With Ready Queue [6]

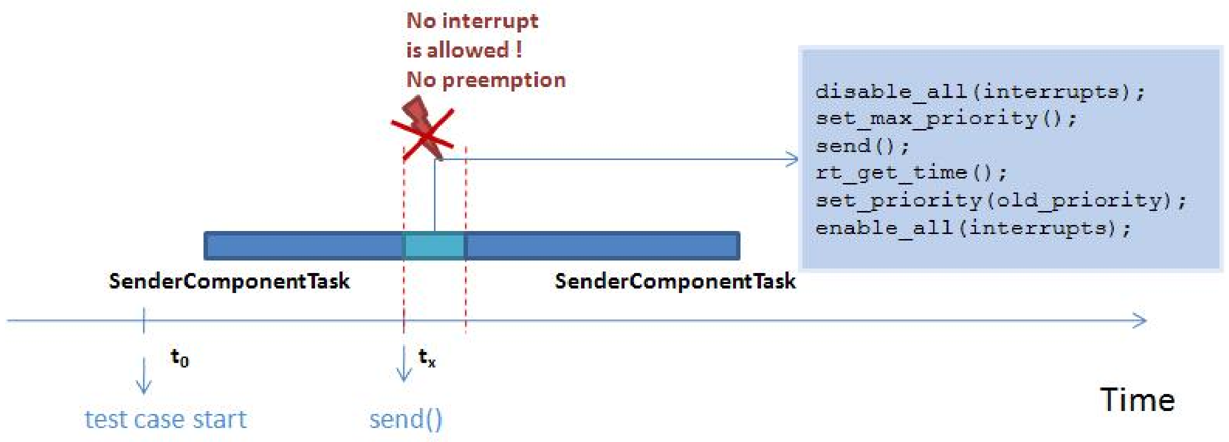
```
testcase SendTimestamp() runs on SenderComponent
{
        float  tx;
    ...
    // other computations
    senderPort.send(msg) -> timestamp tx;
    // other computations
    ...
}
```

LISTING 5.2: Component task with send and timestamp. TTCN-3 code.

5.1.10 Alt Operation With Receive Branches And Break Condition

A more complex interaction is encapsulated into the `alt` statement. One example of one `alt` waiting on two ports is described in Listing 5.3. All the elements described previously are used in this scenario. The proposed solution, with interrupts for events, with handlers, and tasks, represent a real-time alternative to the snapshot semantic proposed in the TTCN-3 standard. [124] demonstrates why snapshot semantic is inefficient and not suitable for real-time applications.

Referring to the example presented in Listing 5.3, in our approach, the `alt` is associated with a task that manages two queues. This is due to the fact that there are two ports



```
disable_all(interrupts);
set_max_priority();
send();
rt_get_time();
set_priority(old_priority);
enable_all(interrupts);
```

FIGURE 5.11: Execution Timeline For A Send With Timestamp [6]

on which the alternative will wait for incoming messages. The queues should be created by the parent component, along with the creation of the Alt task. When a message is received on one of the ports, an interrupt is generated, and a handler for an external event is called. The handler for an external event takes the time stamp and saves it together with the message in one of the abstract queues; the same one that is associated with the port on which the message was received. As manager of the queues, the Alt task takes the last incoming message and compares it against both the time and structural patterns.

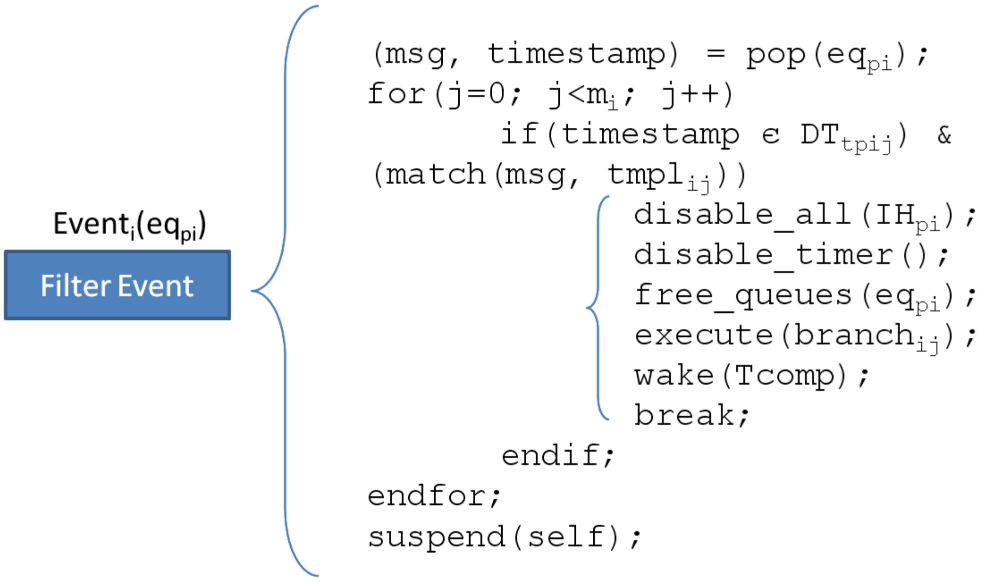
```
testcase InTimeReceive() runs on ReceiverComponent
{
    float tx;
    float ty;
    timespan tz;
    float tt;
    ...
    alt {
        [] pa.receive(tmpla1) within(tx, ty) {
            setverdict (pass);
            }
        [] pb.receive(tmplb1) before tz {
            setverdict (pass);
            }
        [] pa.receive  {
            // any other message,
            // any other time
            setverdict(fail);
            }
        [] pb.receive  {
            // any other message,
            // any other time
            setverdict(fail);
            }
        } break at (tz  + 10*millisec){
            setverdict(error);
            log(...);
            ...
        }
        ...
}
```

Listing 5.3: TTCN-3 code sample for an alt with two ports.

The function Filter presented in Figure 5.12 displays the algorithm for matching. If the message matches, then the behavior associated with that branch is executed next in the context of the current task. If it doesn't match, then the Alt task blocks the waiting of the arrival of another message. The Alt task can be interrupted while executing by using a timer handler or by an I/O interrupt handler, preempted and moved to ready queue. It can also be preempted by tasks with a higher priority.

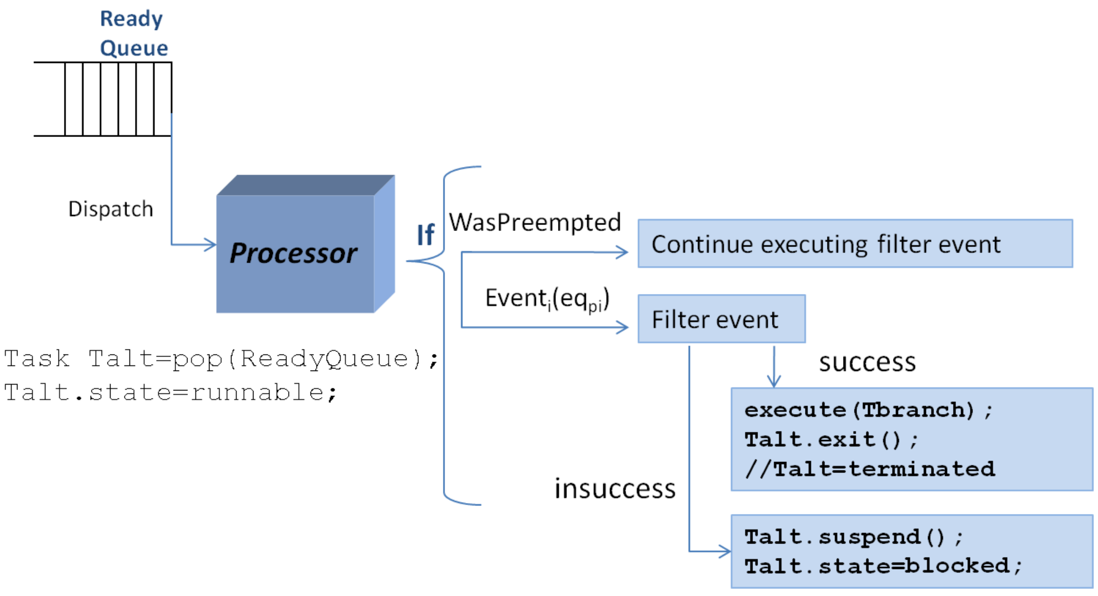
When the processor becomes available and there is no other task with a higher priority running in the system, this then gets the processor and continues its computation (see Figure 5.13).

State transitions for the Alt task are presented in Figure 5.14.



```
(msg, timestamp) = pop(eq_pi);
for(j=0; j<m_i; j++)
        if(timestamp ∈ DT_tpij) &
(match(msg, tmpl_ij))
                disable_all(IH_pi);
                disable_timer();
                free_queues(eq_pi);
                execute(branch_ij);
                wake(Tcomp);
                break;
        endif;
endfor;
suspend(self);
```

FIGURE 5.12: Filter Function After Receiving The Event [6]



```
Task Talt=pop(ReadyQueue);
Talt.state=runnable;
```

```
execute(Tbranch);
Talt.exit();
//Talt=terminated
```

```
Talt.suspend();
Talt.state=blocked;
```

FIGURE 5.13: Behavior Of Alt Task When Running [6]

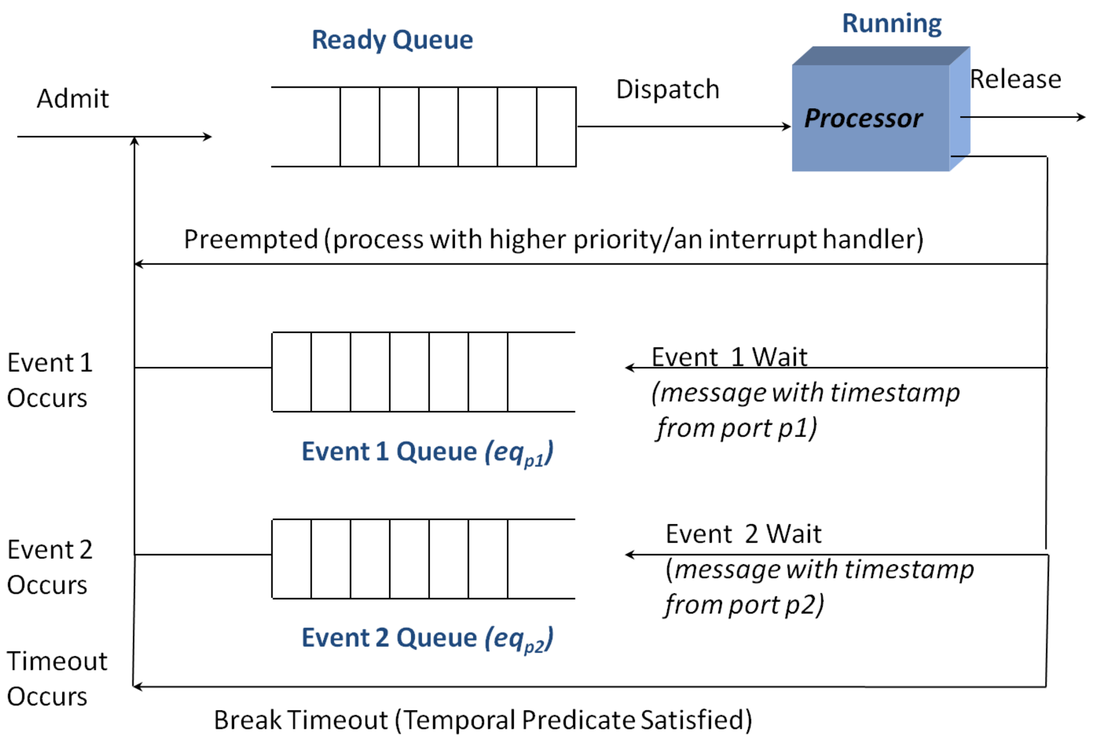
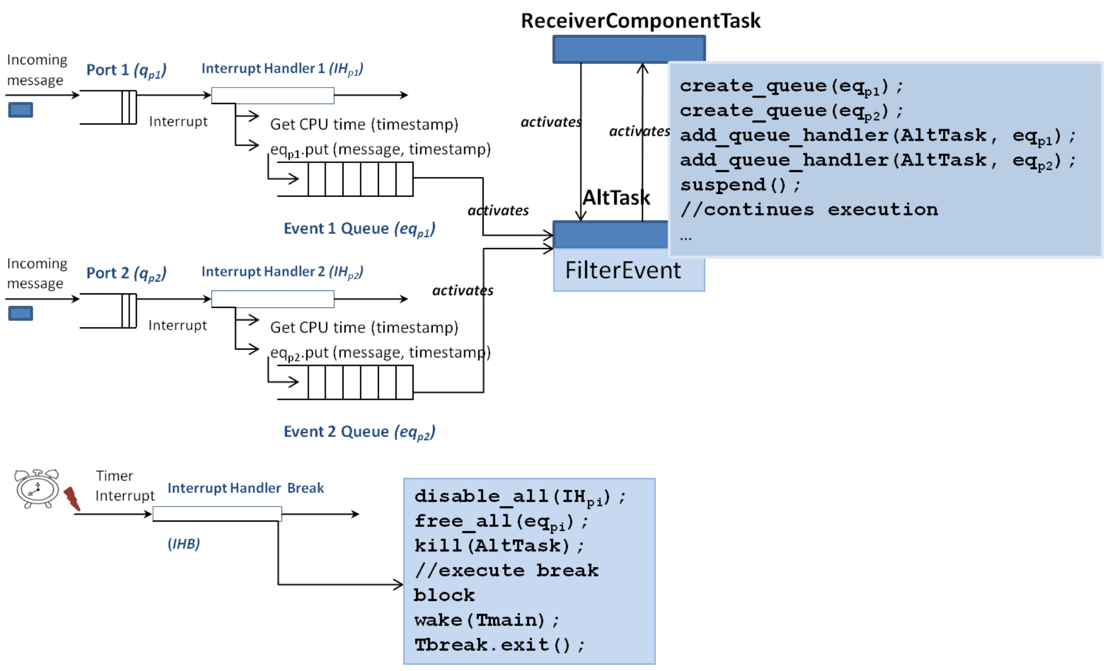

Ready Queue
Running
Admit
Dispatch
Processor
Release
Preempted (process with higher priority/an interrupt handler)
Event 1 Occurs
Event 1 Wait (message with timestamp from port p1)
Event 1 Queue (eq_{p1})
Event 2 Occurs
Event 2 Wait (message with timestamp from port p2)
Event 2 Queue (eq_{p2})
Timeout Occurs
Break Timeout (Temporal Predicate Satisfied)

FIGURE 5.14: Alt Task Transition States [6]

In Figure 5.15 is presented how the alt task interacts with other parts of the system. It is a picture of the participants involved in the `alt` behavior.



```
create_queue(eqp1);
create_queue(eqp2);
add_queue_handler(AltTask, eqp1);
add_queue_handler(AltTask, eqp2);
suspend();
//continues execution
...
```

```
disable_all(IHpi);
free_all(eqpi);
kill(AltTask);
//execute break
block
wake(Tmain);
Tbreak.exit();
```

FIGURE 5.15: Task Interactions For An Alt Statement [6]

Figures 5.16 and 5.17 present two possible execution flows. In the first, two messages arrive, generating interrupts. The Alt task is preempted twice and one of the messages corresponds with one pattern and is received in time. Therefore the execution is continued with the behavior of the associated branch.

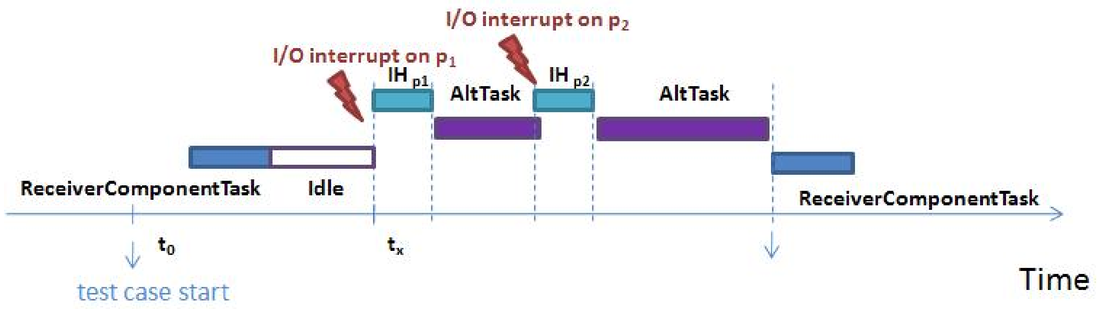


FIGURE 5.16: (a) Possible Flow Of Events [6]

In Figure 5.17 no message is received in time. A timer interrupt is generated after the time interval for the alternative expires and the Alt task is killed. An error behavior will be executed in this situation.

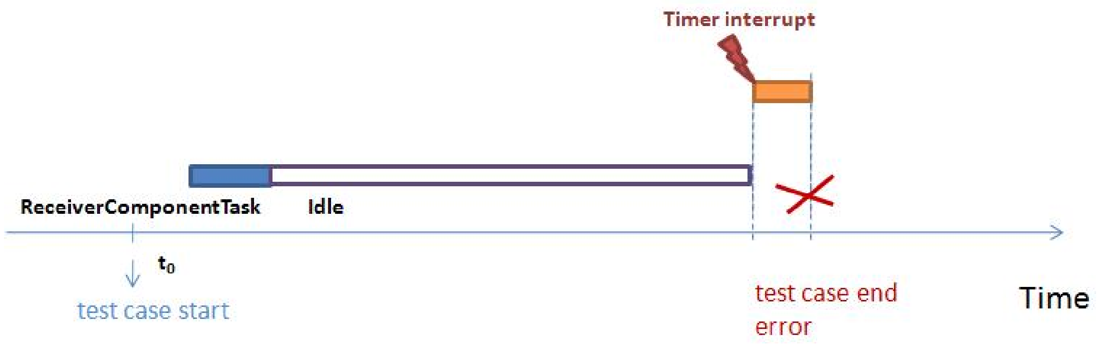


FIGURE 5.17: (b) Possible Flow Of Events [6]

It would be interesting to imagine how the system would look like if we had more than one component. For this, an algorithm for assigning priorities to different tasks associated with different components would be a good choice. The priority should be set in relation to the deadline of each thread. The closer the deadline, the higher the priority. For example, if we have two components, each containing one `alt` statement, we set the priority for the Alt threads as follows: each `alt` is associated with a `break` clause. The time indicated by the temporal predicate of the `break` clause gives the deadline for the Alt task itself. The Alt task with the closest deadline will have the higher priority.

Nevertheless, this heuristic function does not provide the guarantee that the system will respect its deadline. A thorough study of the *SUT* should be performed and other heuristics should be developed.

There are also situations – depending on the malbehavior of the *SUT* – that can induce a malfunctioning of the test system itself. Such a situation would be a flooding of the *TS* with input events. Handling endless incoming events at the highest priority, may lead to deadline overrun on the test system side. These situations should be predicted and taken into consideration during the design phase.

5.2 Real-time Operating System Selection

The market of Operating Systems (OS) is continuously and increasingly developing, as a result of more and more sophisticated requirements. One of the key necessities is to support embedded real-time applications in which the OS must guarantee the timeliness as well as the correctness of the processing. Many OS claim to be Real-time Operating Systems (RTOS), but often only by reviewing the OS specifications, or more detailed information, one can truly identify the OS that enables real-time applications.

The process of selecting the right RTOS is an important and, at the same time, a critical one. It involves knowing all the specifications of different RTOS's in an abundant market of available RTOS's, including from micro kernels to commercial RTOS's.

The design space available to any RTOS is very broad and there are countless sets of characteristics that form the criteria on which the RTOS selection is performed. Selecting the RTOS based on these features is a multidimensional search problem where each dimension corresponds to a RTOS characteristic requiring an exhaustive search, tremendous computing resources and time.

5.2.1 RTOS Candidates

The initial search was performed using the Internet, where a wide variety of RTOSes are available to suit most projects and pocketbooks [122, 123]. Our search revealed 16 RTOSes that required further investigation. The actual number of RTOSes is in fact larger. These were merely the ones that could be applied to our real-time test system.

The first reason for eliminating RTOSes was the *license term conditions*. As with any system software, RTOSes come with various license terms (commercial software, free software, open source software, etc). Our decision was to select only open source RTOSes, even though the more powerful known RTOSes are commercially distributed. The interest in alternatives which are free of charge compared to the conventional, partly expensive RTOS, is increasingly growing. The reasons to apply open source software (OSS) are diverse. Often OSS is selected to compensate for disadvantages of commercial solutions and to use the advantages of OS. These disadvantages can be found in the following areas:

Costs For commercial RTOSes licence costs are to be paid, which compared largely to the implementation costs, are only slightly below or the same. Furthermore, mostly annual maintenance fees incur.

Safety Commercial systems are mostly ”closed source”, which means they cannot be seen in the source code. Thus, possible safety gaps or safety-relevant defective functions cannot be seen in the source code in advance and cannot be corrected.

Stability and Performance Open source systems mostly have a much broader installation basis than commercial systems. They are therefore tested more extensively and optimised through these experiences.

Support In case of commercial RTOSes the support is liable to costs. Mostly because forums or mailing lists for the exchange among each other do not exist.

Table 5.1 presents the eliminated candidates and the main reason for elimination.

Table 5.1: Eliminated RTOS Candidates

RTOS Name	**Vendor**	**Reason for elimination**
QNX	QNX Software Systems	Commercial product
uKOS	uKOS Team	Lack of determinism and/or limited real-time capabilities
VxWorks	Wind River Systems	Commercial product
CapROS	Strawberry Development Group	Limited development information and support
WinCE	Microsoft	Commercial product
Coyotos	Johns Hopkins University’s Systems Research Laboratory	Developed in BitC language
OSE	ENEA	Commercial product
Fiasco	TU Dresden	Limited development information and support
RTLinux	Wind River Systems	Commercial product
C Executive	AMD	Limited portability
scmRTOS	scmRTOS Team	Limited portability

This elimination process left four RTOS candidates (Table 5.2) to be evaluated in detail and ranked on our specific requirements in Section 5.2.2.

5.2.2 RTOS Selection Criteria

Based on the specific requirements of the automotive industry, as well as on the obvious needs for performance, reliability, and cost-effectiveness common to every real-time project, we have divided the selection criteria in two parts: the first part one envelopes

Table 5.2: Remaining RTOS Candidates For Complex Evaluation

RTOS Name	**Vendor**	**Project site**
eCos	Red Hat	www.ecos.sourceware.org
FreeRTOS	Richard Barry & FreeRTOS Team	www.freertos.org
RTAI	RTAI Team	www.rtai.org
RTEMS	Oar Corp.	www.rtems.com

general points of view, such as: supported languages, portability, latest update, commercial status, available API and information about development and support. The second part of the selection process includes more specific features of RTOSes, such as: scheduling algorithms, type of RT (soft of hard), priority levels, kernel ROM size, kernel RAM size, multi-process support, interrupt latency, task switching time, type of IPC mechanism, memory management, task management etc. These two parts of selection criteria are shown in Table 5.3 and Table 5.4 and a detailed argument concerning the elimination criteria for selection is given below, along with a mention that the basic and the first criteria of selecting process is contained in the Section 5.2.1, where the commercial RTOSes and those that don't meet the basic requirements were eliminated.

In order to defend the selection decision we should see a summarized description of each of these RTOSes based on the previous tables:

- **eCos**:

  eCos (embedded Configurable operating system) is an open source, royalty-free, real-time operating system intended for embedded systems and applications, which need only one process with multiple threads. The OS is configurable, and can be customized to precise application requirements, with hundreds of options, delivering the best possible run-time performance and minimized hardware need. eCos

Table 5.3: A General RTOS Selection

	eCos	**FreeRTOS**	**RTAI**	**RTEMS**
Languages Support	Assembly, C, C++, Ada95	C, Assembly	C	C, C++, Ada95
Target CPUs Support	x86, PowerPC, ARM, MIPS, Altera NIOS II, Calm-risc16/32, Freescale 68k ColdFire, Fujitsu FR-V, Hitachi H8, Hitachi SuperH, Matsushita AM3x, NEC V850, SPARC	ARM architecture (ARM7, Cortex-M3) , AVR, AVR32, HCS12, MicroBlaze, MSP430, PIC microcon-troller (PIC18, PIC24, dsPIC), Renesas H8/S, x86, 8052	X86 (with and without FPU and TSC), PowerPC, ARM (StrongARM; ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x), MIPS	ARM, Blackfin, ColdFire, Texas Instruments C3x/C4x DSPs, H8/300, x86, 68K, MIPS, Nios II, PowerPC, SuperH, SPARC
Development Status	eCos 2.0, May 2003	FreeRTOS 4.4.0, July 2007	RTAI 3.5, February 2007	RTEMS 4.6.6, April 2006
Source Model/ License	Open source/ eCos License (GPL with exceptions)	Open source/ Modified GPL	Open source	Open source/ Modified GPL
API	POSIX (1003.1b), ITRON, ”clas-sic/native” API in C and Ada	Well written custom API, based on ”classic” API in C	Custom API derived from RTLinux V1 API	uITRON 3.0 API, POSIX 1003.1b, BSD standards
Development Informa-tion/ Support	Books, papers/ Mailing list	Web tutorials/ Forum	Incomplete documenta-tion/ Web support, mailing list	Wiki/ Contractual support

Table 5.4: A More Specific RTOS Selection

	eCos	**FreeRTOS**	**RTAI**	**RTEMS**
Kernel Architecture	Configurable (monolithic probably)	Microkernel	Module oriented, loadable to Linux kernel	Monolithic
Multi-process Support	Yes	Yes	Yes	No
Scheduling Algorithm	Preemptive: bitmap/ multi-level queue. Unique priorities/ timeslice.	Preemptive/ cooperative. Highest priority first.	Fully preemptive, Round Robin, timeslice, fixed priority, dynamic priorities, coopertative multitasking	Rate-monotonic scheduling. Configurable: non/preemp-tive, no/timeslice.
Priorities interval	0 .. 31	Configurable	0x3fffFfff .. 0	1 .. 255
Task States	Ready,Running, Suspended, Sleep,Exited	Running,Ready, Blocked, Suspended	Ready,Running, Suspended, Delayed, Semaphore, Send,Receive, Rpc,Return, Deleted	Executing, Ready,Blocked, Dormant, Non-existent
Max. Number of Tasks	Configurable	Unlimited	Unlimited	Configurable
Co-routine Impl.	No	Yes	No	No
Virt. Mem. Support	No	No	Yes	No
Pre-emptable ISR	No	Yes	Yes	Configurable

has been designed to support applications with real-time requirements, providing features such as full preemptability, minimal interrupt latencies, and all the necessary synchronization primitives, scheduling policies, and interrupt handling mechanisms needed for this type of application. eCos also provides all the functionality required for general embedded application support including device drivers, memory management, exception handling, C, math libraries, etc. It is programmed in the C programming language, and has compatibility layers and APIs for POSIX and ITRON. eCos was designed for devices with memory size in the tens, up to hundreds of kilobytes, or with real-time requirements. It can be used on hardware with too little RAM to support embedded Linux, which currently needs a minimum of about 2 MB of RAM, not including application and service needs.

eCos runs on a wide variety of hardware platforms, including ARM, CalmRISC, FR-V, Hitachi H8, IA-32, Motorola 68000, Matsushita AM3x, MIPS, NEC V8xx, Nios II, PowerPC, SPARC, and SuperH [117].

- **FreeRTOS**:

  FreeRTOs is a real-time operating system for embedded devices, being ported to several microcontrollers. The FreeRTOS scheduler is designed to be small and simple. It can be configured for both preemptive or cooperative operation. To make the code readable, easy to port, and maintainable, it is written mostly in C, but there are a few assembler functions included where needed. FreeRTOS allows an application to use coroutines (a lightweight task that uses very little memory.), as well as tasks. The FreeRTOS kernel itself is comprised of only three or four C files, depending on whether coroutines are used or not. The FreeRTOS.org site contains RTOS tutorials, details of the RTOS design and performance comparison results for various microcontrollers, and the distribution comes with prepared configurations and demonstrations for every port, allowing rapid application design.

  Supported architectures: ARM architecture ARM7 (ARM Cortex-M3), AVR, AVR32, HCS12, MicroBlaze, MSP430, PIC microcontroller PIC18, PIC24, dsPIC, Renesas H8/S, x86, 8052 [115].

- **RTAI**:

  RTAI stands for Real-time Application Interface. It is a real-time extension for the Linux kernel, which allows for the possibility to develop applications with strict timing constraints for Linux. RTAI provides a deterministic response to interrupts, POSIX compliant and native RTAI realtime tasks. Realtime Application Interface consists mainly of two parts: a patch to the Linux kernel which introduces a hardware abstraction layer and a broad variety of services required for general real-time applications.

RTAI supports several architectures:x86 (with and without FPU and TSC), PowerPC, ARM (StrongARM; ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x), MIPS [116].

- **RTEMS**:

  RTEMS (Real-time Executive for Multiprocessor Systems) is a free open source real-time operating system designed for embedded systems. RTEMS is designed to support various open API standards including POSIX and uITRON. The API now known as the Classic RTEMS API was originally based upon the Real-time Interface Executive Definition (RTEID) specification. RTEMS does not provide any form of memory management or processes. In POSIX terminology, it implements a single process, multithreaded environment. This is reflected by the fact that RTEMS provides nearly all POSIX services apart from those which are related to memory mapping, process forking, or shared memory. RTEMS includes a port of the FreeBSD TCP/IP stack as well as support for various filesystems including NFS and the FAT filesystem.

  RTEMS is designed for real-time, embedded systems and has been ported to various target processor architectures: ARM, Blackfin, ColdFire, Texas Instruments C3x/C4x DSPs, H8/300, i386, Pentium, and above members of the X86 architecture, 68K, MIPS, Nios II, PowerPC, SuperH, SPARC [118].

5.2.3 Evaluation Results

Based on specific requirements and principles, we decided that the most suitable of our real-time application were the FreeRTOS and RTAI operating systems.

A reason to renounce eCos was that the latest stable verion was in May, 2003 and has remained frozen since then. An option to get most up-to-date version is by using the CVS, however, a less tested version of eCos.

The argument for choosing RTAI was that its documentation is well written, and it provides a complete API with very flexible features, appropriate for designing a great range of real-time application. On the other hand, FreeRTOS is a very small, simple and concise operating system, making it suitable for small applications on small platforms. It was decided to try to implement our concepts on these two platforms, one more general purpose real-time and the other more small and specific.

Since in both cases the majority of the code is written in C language, it is highly portable and able to be ported to many physical platforms. A strong advantage of FreeRTOS and RTAI is that for each supported platform, the code includes a demo project demonstrating how to use the code on that specific platform. Unfortunately, this feature was not found on the other systems, where installation, configuration and development required more effort. FreeRTOS's strength is its small size, making it possible to run where most other operating systems would not fit. RTAI's strength is its flexibility. Both FreeRTOS and RTAI will be presented in the following two sections.

5.3 FreeRTOS. Important Features.

Simple, Portable, Concise! This is the design philosophy of FreeRTOS. Therefore, the kernel of FreeRTOS is small, comprised of only three or four files. To make the code readable, easy to port, and maintainable, it is written mostly in C, but there are a few assembler functions included where needed to have an increase in performance.

Preemptive/Cooperative Scheduling. FreeRTOS relies on a scheduler that can be configured to both *preemptive* and *cooperative* scheduling policies and allows the usage of *coroutines* (lightweight tasks that use very little memory), as well as the usage of *tasks*.

Time Measured By Tick Count. The FreeRTOS real-time kernel measures time using a **tick** count variable. A timer interrupt (the RTOS **tick interrupt**) increments the tick count with strict temporal accuracy - allowing the real-time kernel to measure time to a resolution of the chosen timer interrupt frequency. The tick count is used by the scheduler, so that each time the tick count is incremented, the real-time kernel must check to see if it is now time to unblock or wake a task. An useful function to get the count of ticks since the scheduler was started is the *xTaskGetTickCount()* function. This function represents the correspondent, at the RTOS level, of the `now` instruction from Real-time TTCN-3.

Tick Hook Function. A tick hook function is a function that executes during each RTOS tick interrupt. It can be used to optionally execute application code during each tick ISR.

Priority-based Scheduling. The scheduling decision is made based on priorities of the tasks: task given processing time will always be the highest priority task that is able to run. Task priorities are denoted by numbers, where low numbers denote low priority tasks.

Task Delay. Among other utilities that FreeRTOS provides to control tasks, there are also *vTaskDelay()* and *vTaskDelayUntil()* which are used when there is a necessity to block a task for a given number of ticks. *vTaskDelay()* represents the correspondent, at the RTOS level, of the `wait` instruction from Real-time TTCN-3.

5.4 RTAI. Important Features.

Between Linux And Hardware. The design idea of RTAI is that it adds a layer between the Linux kernel and the hardware. The RTAI kernel manages real-time tasks according to their priorities. In the RTAI context, the Linux kernel is also regarded as a real-time task, holding the lowest priority. Thus, all non real-time interrupts are handed out to the Linux kernel.

Schedulers And Hard Real-time In Kernel And In User Space. The scheduler is the heart of RTAI and it provides its real-time capabilities. RTAI provides symmetric hard real-time services within kernel or user space. The two schedulers that can operate in both user and kernel mode are responsible with that support. They differ only in relation to the objects they can schedule. One is simply a GNU/Linux co-scheduler which supports hard real-time for all Linux scheduleable objects like *processes*, *threads*, or *kthreads*. The other supports real-time as well as for RTAI own kernel tasks. Scheduler has different modes, as: FIFO in Native mode or EDF.

Periodic And One-Shot Timers. RTAI manages the timer in two distinct ways: periodic and absolute or “one-shot” modes. In *periodic mode* the timer is set to interrupt at a fixed, non varying, period. There are no updates to the timer from the scheduler. Periodic mode is available to reduce the significant overhead required to program the timer registers. In *one-shot mode* the timer is reprogrammed every time the scheduler is called. There is an overhead involved, but it has better time resolution and more adaptable scheduling. The timer generates an external interrupt like any other interrupt source in the system. It is handled in a specific way, because it plays a central role in the computer system, in the same way the RTOS uses the timer interrupts as scheduling triggers.

Interrupt Handling. In a hard real-time system, internal and external interrupts must be served in a well bounded time. Since interrupts are usually served with interrupts disabled, the ISR length must be as short as possible.

RTAI Modules. RTAI is a more complex RTOS than FreeRTOS and it consists of several modules. The modules with the needed RTAI capabilities must be loaded, before usage. The most important modules are:

The *core module of RTAI*, through which Linux works toward the hardware is filtered, making RTAI the only master of the hardware.

The *real-time scheduler module* distributes the CPU to different tasks; there are three different schedulers:

UP : only for uniprocessors; the process with the highest priority gets the CPU.

SMP : for multiprocessors; it can schedule tasks to more than one CPU and it's a priority driven scheduler.

MUP : only for multiprocessors; MUP scheduler views a multiprocessor machine as a collection of many uniprocessors and each CPU can have its timer programmed differently.

5.5 Summary

This chapter has presented the general architecture for the real-time testing framework developed in this thesis. It has also shown the differences between an implementation design for a classical TTCN-3 platform and an implementation design for a real-time TTCN-3 platform. Design patterns for realizing the real-time concepts presented in Chapter 4 have been derived from our experiences with implementing these concepts on two concrete real-time operating systems: FreeRTOS and RTAI. These operating systems were selected from among others, based on several criteria, such as: supported languages, portability, latest update, commercial status, supported features. FreeRTOS is small and minimal, specialized on small embedded systems, while RTAI is complex and more suitable for general purpose, but nevertheless both of them feature hard real-time capabilities and flexible APIs. Being different both in purpose and in the size of their API, we considered it a challenge to realize our concepts on both these platforms. Firstly, to prove that the set of real-time extensions proposed here can be realized with minimal basic services (the case of FreeRTOS), and secondly, to prove that even on a more complex featured RTOS, a time-bounded realization for these concepts is still possible.

Chapter 6

Mappings For The Real-time Test Concepts

"Have no fear of perfection, you'll never reach it."
Salvador Dali

In this chapter we are going to present the mappings of the real-time concepts for TTCN-3 on the two real-time platforms selected at the end of Chapter 4: FreeRTOS and RTAI. These operating systems were selected among others, based on several criteria, such as: supported languages, portability, latest update, commercial status, supported features. FreeRTOS is small and minimal, specialized for small embedded systems, while RTAI is complex and used more for general purpose, although both feature hard real-time capabilities and flexible APIs. Being different in purpose and size of their API, we considered it a challenge to realize our concepts on both of these platforms. Firstly to prove that the set of real-time extensions proposed here can be realized with minimal basic services (the case of FreeRTOS), and secondly to prove that even on a more complex featured RTOS a time-bounded realization for those concepts is still possible.

6.1 Linux With RTAI Based Real-time Testing Framework

This is the section where we present the mapping of the real-time concepts, described from a theoretical perspective in Chapter 4, on a real-time operating system platform, namely RTAI. All the implementation examples featured in this chapter, show these mappings, and follow logically the architectural paradigms from Chapter 5; these paradigms are realized using a means that is specific to a real-time Linux-based platform. In our case, the chosen platform is a RTAI patched version of Linux, RTAI application interface for real-time having already been introduced in Section 5.4 of the previous chapter.

All the examples shown and discussed in the following sections, are implemented on an Ubuntu 8.04 - Hardy Heron distribution, released in April 2008. On this distribution, a vanilla kernel version 2.6.24 had been patched with RTAI patch version 3.6-cv, and compiled and installed according to the predefined installation procedure. For implementing network communication on sockets, as presented in code snippets 6.26 and 6.30, RTnet version 0.9.12 [119] had to be installed, and the used kernel had to be recompiled without network drivers.

6.1.1 *TS* As A Real-time Kernel Module

The module structure of the real-time kernel can be seen as the main frame into which each individual real-time concept will be implemented. Thus, the *TS* itself is going to be mapped to a kernel module. We chose this approach, realizing our concepts in kernel space - despite it being more difficult by means of debugging and development - because of its increased performance regarding timing aspects (a few microseconds) as opposed to a user space implementation [116].

Kernel modules must have at least two functions: a 'start' function, used for initializations, known as `init_module()` which is requested when the module is inserted into the kernel (`insmod`), and an 'end' function, used for cleaning up used handlers and references, known as `cleanup_module()` which is requested just before it is removed (`rmmod`).

Typically, `init_module()` either registers a handler for some used resource with the kernel, or it replaces one of the kernel functions with its own code. The `cleanup_module()` function is supposed to undo whatever `init_module()` has done, so the module can be unloaded safely [139]. In our case, `init_module()` starts the real-time scheduler by calling `rt_set_oneshot_mode()` and `start_rt_timer()` functions, while the `cleanup_module()` deactivates the real-time scheduler, by invoking `stop_rt_timer()` method.

`rt_set_oneshot_mode()` function sets the oneshot mode for the timer. This allows further defined tasks to be timed arbitrarily, based on the cpu clock frequency. This method marks the starting point of the real-time execution and it must be requested before using any time related function, including time conversions. With a call to `start_rt_timer()` the real-time clock is actually activated and `stop_rt_timer` sets the timer back into its default mode.

6.1.2 Real-time Test Case With Special Operations

The next example – referencing the Listings 6.1, 6.2 and 6.3 – shows how a test case can be implemented using our real-time approach for RTAI. More precisely, the excerpts of code presented in Listings 6.2 and 6.3 are realizing the abstract RT-TTCN-3 specification from Listings 6.1. A mapping pattern can be deduced by comparing the two perspectives. As a visual convention, used for making up the difference between the two perspectives – RT-TTCN-3 perspective and RTAI perspective – we chose different background coloring and different fonts for the code samples: light-grey background for the RT-TTCN-3 specification and larger fonts versus white background for the RTAI module.

Listing 6.1 shows the definition of a test case named `DemoTestCase()`. This is set to run on the component of type `MTCComponentType`, previously defined to contain, among others, an integer constant representing the component's id. The body of the test case

```
type component MTCComponentType{                                 1
    const integer mtcID := 1;                                    2
    ...                                                          3
    // other definitions                                         4
}                                                                5
// This is the function that contains the test behavior          6
testcase DemoTestCase() runs on MTCComponentType{                7
    var timespan ts := 10*millisec;                              8
    var float x;                                                 9
    var float y;                                                10
    ...                                                         11
    // Ask for the current time                                 12
    x := now;                                                   13
    ...                                                         14
    // Delay the execution for 10 milliseconds                  15
    wait(ts);                                                   16
    ...                                                         17
    y := testcasestart;                                         18
    ...                                                         19
}                                                               20
...                                                             21
// Here is started the execution of the test case               22
control{                                                        23
    execute(DemoTestCase());                                    24
    ...                                                         25
    // more test cases might follow                             26
}                                                               27
```

Listing 6.1: A Testcase Example with RT-TTCN-3

– extended in Listing 6.1, Lines 8-19 – contains also some of the newly introduced real-time concepts, as `now`, `wait` and `testcasestart`. In Listing 6.1, Line 24 the test case function is launched to execution by using the `execute` construct from TTCN-3. When a test component becomes active, a component of type `MTCComponentType` will be automatically created, this being the main test component (MTC) on which the behavior of the test case is going to be executed.

Creating a real-time test case means starting a real-time task associated with the MTC. Listing 6.2 presents the excerpt of code where the task routine implementing the behavior of the test case is defined as `testcase_function`. Together with the routine definition, there are two more structures associated with the test case: `tc_task` structure, being handler of the test case task routine, and `tc_struct` keeping additional information related to the test case, e.g. test case's start time, etc..

We can observe in the body of `testcase_function`, that the first thing that happens after entering the function is recording the start time by invoking method `rt_get_time_ns()` and saving the value into the `tc_struct` associated with this test case (Line 18). A mapping for the `now` operation from RT-TTCN-3 occurs in Line 20; the current time value is obtained by using again the library function `rt_get_time_ns()`. This function returns the time in nanoseconds since the `start_rt_timer()` was called. A mapping for the `wait` operation from RT-TTCN-3 is done in Line 23-26, by invoking the method `rt_sleep()`. A call to this function suspends execution of the caller task for a time of '`delay`' internal count units, where '`delay`' is being given as an input parameter of this function. During this time, the CPU is used by other tasks. For the transforming of nanoseconds into internal count units, conversion function `nano2count` is being used. The count units are related to the time base that was set when real-time clock was started (e.g. using `rt_set_oneshot_mode()` or `rt_set_periodic_mode()` time scalings).

```
 1
 2  /*
 3  ** Real-time Test Case or Real-time Main Test Component.
 4  **/
 5  ...
 6  /* we'll fill this in with our main test component task */
 7  static RT_TASK tc_task;
 8  /* we'll keep here all the information related to the test case */
 9  static TC_STRUCT tc_struct[0];
10
11  /*** This method is implementing the behavior for the
12  * test case */
13  static void testcase_function(long tc_id) {
14      RTIME now, delay;
15      /*** Here I write the behavior of the real-time test case */
16      /*** Recording the time when the test case started it's
17       * execution */
18      tc_struct[0].tc_start_timestamp = rt_get_time_ns();
19      /*** Reading the real-time clock */
20      now = rt_get_time_ns();
21      /*** Transforming a delay of 10 milliseconds in internal
22       * counts values. */
23     delay = nano2count(10000000);
24      /** The task will be blocked waiting for the 'delay'
25       * time to expire. */
26     rt_sleep(delay);
27      ...
28      /*** Recording the time when the test case stopped */
29      tc_struct[0].tc_stop_timestamp = rt_get_time_ns();
30          return;
31  }
```

Listing 6.2: Simple Real-time TestCase: Task Function

Listing 6.3 shows how the real-time task associated with the test case is initialized in the `init_module()` section, as well as how is it launched for execution. Task's initialization is realized in Lines 7-20 with method `rt_task_init()`. This creates a new real-time task. Each parameter of the function is explained in the comments which accompany the code. The newly created real-time task is initially in a suspend state. It can be made active by calling: `rt_task_make_periodic`, `rt_task_make_periodic_relative_ns`, or `rt_task_resume` which is actually invoked in Line 23. In the `cleanup_module()`

section the task structure associated with the real-time task is deleted and the real-time scheduler is stopped.

```

int init_module(void) {
        rt_set_oneshot_mode();
        (void)start_rt_timer(1);
        /* initialize the task */

        rt_task_init( /*** Pointer to the task structure */
                  &tc_task,
                  /*** The actual task function */
                  testcase_function,
                  /*** Initial task parameter */
                  tc_struct[0].tc_id,
                  /*** 1K stack */
                  1024,
                  /*** Priority of the task */
                  RT_SCHED_LOWEST_PRIORITY + 2,
                  /*** Don't use floating point */
                  0,
                  /*** Don't use signal handler */
                  0);

        /* make the task ready to run */
        rt_task_resume(&tc_task);
        return 0;
}

void cleanup_module(void){
    rt_task_delete(&tc_task);
    stop_rt_timer();
    return;
}
```

Listing 6.3: Simple Real-time TestCase: Init/Cleanup Module Functions

The information presented in this section is summarized in Table 6.1, where the synthesized code mappings between RT-TTCN-3 abstract concepts and the key implementation aspects in C, using RTAI API, are displayed. The illustrated RT-TTCN-3 abstract concepts are: `now`, `wait`, `testcasestart` and `testcase`.

6.1.3 Real-time Test Component

This section discusses the way parallel test components (PTCs) are being handled in real-time. In addition to the mechanisms of creating and starting test components, an accurate recording of starting and stopping execution time is also implicitly performed.

Listing 6.4 presents a classical TTCN-3 specification excerpt where, inside a test case function, a parallel test component(PTC) is created and then started. Lines 46-48

TABLE 6.1: RT-TTCN-3 To RTAI Mappings. Part I.

RT-TTCN-3 Concepts	**RTAI C Code**
`now`	`rt_get_time_ns()`

`var timespan ts := 10*millisec;`
`wait(ts);`
	`RTIME delay;`
	`delay = nano2count(10000000);`
	`rt_sleep(delay);`

`var float y;`
`y := testcasestart;`
	`static TC_STRUCT tc_struct [0];`
	`static void testcase_function(long tc_id){`
	`       tc_struct[0].tc_start_timestamp =`
	`           rt_get_time_ns();`
	`       ...`
	`       return;`
	`}`

`type component MTCComponentTypef{`
`     const integer mtcID := 1;`
`}`
`testcase DemoTestCase()runs on`
`         MTCComponentTypef{`
`     ...`
`}`
`control{`
`     execute(DemoTestCase());`
`     ...`
`}`
	`static RT_TASK tc_task;`
	`static TC_STRUCT tc_struct[0];`
	`static void testcase_function(long tc_id){`
	`     ...`
	`}`
	`int init_module(void){`
	`     ...`
	`     rt_task_init(&tc_task,`
	`             testcase_function,`
	`             tc_struct[0].tc_id,`
	`             1024,`
	`             RT_SCHED_LOWEST_PRIORITY+2,`
	`             0,`
	`             0);`
	`     rt_task_resume(&tc_task);`
	`     ...`
	`     return 0;`
	`}`
	`void cleanup_module(void){`
	`     rt_task_delete(&tc_task);`
	`     ...`
	`     return;`
	`}`

```
28 type component MTCComponentType{
29     const integer mtcID := 1;
30     ...
31     // other definitions
32 }
33 type component PTCComponentType{
34     const integer ptcID := 2;
35     ...
36     // other definitions
37 }
38 // Function that contains the behavior of the test component
39 function BehaviorTestComponent() runs on PTCComponentType{
40     // PTC's Behavior
41     ...
42 }
43 // This is the function that contains the test behavior
44 testcase DemoTestCase() runs on MTCComponentType{
45     var PTCComponentType NewPTC;
46     NewPTC := PTCComponentType.create;
47     ...
48     NewPTC.start(BehaviorTestComponent());
49     ...
50 }
51 ...
52 // Here is started the execution of the test case
53 control{
54     execute(DemoTestCase());
55     ...
56     // more test cases might follow
57 }
```

LISTING 6.4: An Example with a RT-TTCN-3 Test Component

present these two operations. The behavior executed by the newly created PTC is the same one specified by the function defined on Line 39, and the structural information associated with the component is also given by the component type defined on Line 39.

There are similarities in the way a test case and a test component are implemented. They are both transformed into real-time kernel tasks which are created and then started using the kernel routines: in Listing 6.7, Lines 3-12 it can be observed how both test case task and test component task are created using the `rt_task_init()` function in the `module_init` section. When a task is initialized, a certain priority is assign to it. The priority scale in RTAI ranges between predefined constants `RT_SCHED_HIGHEST_PRIORITY` and `RT_SCHED_LOWEST_PRIORITY`, with `RT_SCHED_HIGHEST_PRIORITY` < `RT_SCHED_LOWEST_PRIORITY`. In our example, the test component gets a lower priority than the test case task. In the `cleanup_module` section, both task handlers, the one associated with test case task and the one associated with the test component task, are deleted (Lines 27-28).

The test component function, `comp_function`, that is referred to at the initialization of the component task, Listing 6.5, Lines 12, is illustrated in Listing 6.5, together with

```
 1              /*** Creating and Starting a Test Component ***/
 2  ...
 3  RT_TASK tc_task;
 4  RT_TASK comp_task;
 5  RTIME testcomponentstart, testcomponentstop;
 6  int tc_id = 1;
 7  int comp_id = 2;
 8  bool alive = true;
 9
10  /***  This is the function that realizes the behavior
11   * of the test component */
12  void comp_function(long comp_id) {
13      /*** Record the starting time of the test component
14       * at the beginning of the component's routine */
15      testcomponentstart = rt_get_time_ns();
16      while(alive){
17          /*** Here is the behavior of the test component */
18          ...
19      }
20      /*** Record the stopping time of the test component
21       * at the end of the component's routine */
22      testcomponentstop = rt_get_time_ns();
23      return;
24  }
```

Listing 6.5: Creating and starting a test component: Test Component Task Function

the task structures for test case, for test component, and other variables containing information related to them. Also, in Listing 6.5, inside the body of the function that describes the behavior of the test component, time stamp variables are used to store the time of the beginning and the ending of the execution of the test component. These times are obtained by invoking the `rt_get_time_ns()` library function, provided by RTAI's API (Lines 15-22).

```
1  void testcase_function(long tc_id) {
2      /*** Here starts the behavior of the test case */
3      ...
4      /*** Starting test component task */
5      rt_task_resume(&comp_task);
6      ...
7      return;
8  }
```

Listing 6.6: Creating and starting a test component: Testcase Task Function

The main difference between a test case task and a test component task is that the former is activated directly from the `init_module` section (see Listing 6.7, Line 27), while the latter is activated from the test case context in Listing 6.6, Line 5. Depending on the context where it is started in the TTCN-3 specification, the test component task can be also activated from another test component's function definition.

```
int init_module(void) {
        /*** Initialize the test case task, aka main test component */
    rt_task_init( &tc_task,
                  testcase_function,
                  tc_id,
                  1024,
                  RT_SCHED_LOWEST_PRIORITY - 3,
                  0,
                  0);
    /*** Initialize the test component that is going to be started
     * from within the test case */
    rt_task_init( &comp_task,
                  comp_function,
                  comp_id,
                  1024,
                  RT_SCHED_LOWEST_PRIORITY - 2,
                  0,
                  0);

    /*** Make the tasks ready to run.....
     * test case aka. main component task */
    rt_task_resume(&tc_task);
        return 0;
}

void cleanup_module(void){
    rt_task_delete(&tc_task);
    rt_task_delete(&comp_task);
    ...
    return;
}
```

Listing 6.7: Creating and starting a test component: Init/Cleanup Module Functions

Table 6.2 presents the schematic mappings between abstract RT-TTCN-3 concepts and their implementations in RTAI. The important concepts presented in this section are: `testcomponentstart`, `testcomponentstop`, component `create` and component `start` operations.

6.1.4 Starting And Stopping A Test Component At A Given Time

Even more real-time semantic flavor can be added to the test components, by allowing the tester to specify the time at which one test component should be starting its execution and the time at which this execution should be stopped. In Listing 6.8 the precise timings for starting and stoping a component are associated to the `start`, and respective `stop` operations in Lines 65-68. The illustrated `start` operation of the PTC is scheduled to take place after `100*microseconds` relative to the current time point of test case's execution. The line that follows, specifies that the execution of the PTC, who's starting point was previously planned, should be stopped after `200*microseconds`, calculated relative to the current time point of the test case's execution.

Table 6.2: RT-TTCN-3 To RTAI Mappings. Part II.

RT-TTCN-3 Concepts	**RTAI C Code**
testcomponentstart	rt_get_time_ns()
testcomponentstop	rt_get_time_ns()

type component PTCComponentType{
     const integer ptcID := 2;
     ...
}
function BehaviorTestComponent()
     runs on PTCComponentTypef{
     // PTC's Behavior
     ...
}
...
NewPTC := PTCComponentType.create;
...
	RT_TASK comp_task;
	int comp_id = 2;
	void comp_function(long comp_id){
	     ...
	     return;
	}
	int init_module(void){
	     ...
	     rt_task_init(&comp_task,
	                                        comp_function,
	                                        comp_id,
	                                        1024,
	                                        RT_SCHED_LOWEST_PRIORITY-2,
	                                        0,
	                                        0);
	     ...
	     return 0;
	}
	void cleanup_module(void){
	     rt_task_delete(&comp_task);
	     ...
	     return;
	}

NewPTC.start(BehaviorTestComponent())	rt_task_resume(&comp_task)

```
...
// This is the function that contains the test behavior
testcase DemoTestCase() runs on MTCComponentType{
    var PTCComponentType NewPTC;
    NewPTC := PTCComponentType.create;
    ...
    // The PTC should be started after 100*microseconds
    NewPTC.start(BehaviorTestComponent())
                at (now + 100*microsec);
     // The PTC should be stopped after 200*microseconds
    NewPTC.stop at (now + 200*microsec);
    ...
}
```

Listing 6.8: RT-TTCN-3 Test Component Start/Stop At

The RT-TTCN-3 specification from Listing 6.8 has the associated implementation presented in its key points in the code snippet from Listings 6.9-6.11.

```

/*** This function comes in action at a fixed  time point and
 * kills the associated PTC task */
void comp_stop_function(long comp_stop_id) {
    /*** Records the time when the test component is suspended
     * and killed */
    timestamp_compstop = rt_get_time_ns_cpuid(cpuid_comp);
    comp_struct[0].comp_stop_timestamp = timestamp_compstop;
    /*** The associated PTC task is suspended here */
    rt_task_suspend(&comp_task);
    /*** The associated PTC task is deleted here */
    rt_task_delete(&comp_task);
    return;
}
```

Listing 6.9: Starting and Stopping a Test Component at a Given Time: Component Stop Task Function.

The PTC is initialized in the `module_init` section – Listing 6.10, Line 13 – in the same way as in the previous section. Together with the real-time task for the PTC, a task associated with the test case is also initialized in Line 4. The third initialization is for a task whose purpose is to stop the execution of the PTC when the time comes. Because this task should be able to interrupt and preempt the PTC whenever it's required, it is created with a priority higher than that of the associated PTC: `RT_SCHED_LOWEST_PRIORITY-2` < `RT_SCHED_LOWEST_PRIORITY-3`. The component structure associated with the stop task is: `comp_stop_task` and its behavior is defined in the `comp_stop_function` which is presented in Listing 6.9. In the `cleanup_module`, Line 36, the task associated with the `comp_stop_task` is deleted.

We may now generalize the problem and take into account a test system with more than one PTC. Some of the PTC are going to be scheduled to stop at certain time points.

```
1  ...
2  // ~ 100 micro sec
3  #define DELAYSTART 100000
4  // ~ 200 micro sec
5  #define DELAYSTOP 200000
6  ...
7  void testcase_function(long tc_id) {
8      ...
9      /*** The PTC task is activated here.*/
10     rt_task_make_periodic_relative_ns(
11         /*** Task structure associated with the PTC task */
12         &comp_task,
13         /*** Activation time for the stop task */
14         DELAYSTART,
15         /*** Period */
16         0);
17     /*** The task for stopping the PTC is activated here.*/
18     rt_task_make_periodic_relative_ns(
19         &comp_stop_task,
20         DELAYSTOP,
21         0);
22     ...
23     return;
24 }
```

Listing 6.10: Starting and Stopping a Test Component at a Given Time: Testcase Task Function.

Therefore, there should be one stop task routine for each PTC, taking care to stop the correspondent PTC when the time comes.

Listing 6.10 presents the body of the test case task function, with the emphasis on the part where the component task and the stop task are being activated. The task activations are realized by the library function `rt_task_make_periodic_relative_ns()`. This method marks the tasks that were previously created with `rt_task_init`, as suitable for a periodic execution, using the period indicated by the last parameter of the method, when `rt_task_wait_period` is called. In our case this input parameter is set to 0, as we don't target a periodical execution (see Lines 16, 21). The time for starting the test component is given by the `DELAYSTART` parameter at the activation of `comp_task` (see Line 14); and the time for stopping the test component is given by the `DELAYSTOP` parameter at the activation function of `comp_stop_task` (see Line 14). `DELAYSTART` and `DELAYSTOP` values are relative to the current time and they represent nanoseconds.

Listing 6.9 presents the body of the stop task function. In Lines 7-8 the current time is asked and then saved for the purpose of being associated with the time when the test component had been stopped. The library method `rt_task_suspend()` is invoked in Line 10 to mark the component task as suspended. The component task is then deleted by calling the `rt_task_delete()` function, in Line 12.

```
int init_module(void) {
    ...
    /*** Initialization of MTC task */
    rt_task_init( &tc_task,
                  testcase_function,
                  tc_id,
                  1024,
                  RT_SCHED_LOWEST_PRIORITY - 3,
                  0,
                  0);

    /*** Initialization of PTC task */
    rt_task_init( &comp_task,
                  comp_function,
                  comp_id,
                  1024,
                  RT_SCHED_LOWEST_PRIORITY - 2,
                  0,
                  0);
    /*** Initialization of task that stops the PTC at a given time. */
    rt_task_init( &comp_stop_task,
                  comp_stop_function,
                  comp_stop_id,
                  1024,
                  RT_SCHED_LOWEST_PRIORITY - 3,
                  0,
                  0);

    /*** MTC is activated here. Testcase is started... */
    rt_task_resume(&tc_task);
        return 0;
}

void cleanup_module(void){
    rt_task_delete(&tc_task);
    rt_task_delete(&comp_stop_task);
    ...
    return;
}
```

Listing 6.11: Starting and Stopping a Test Component at a Given Time:Init/Cleanup Module

The concepts presented in this section and their implementations are shortly summarized as mappings in Table 6.3.

6.1.5 Starting And Stopping A Test Component At A Given Time Using Timer Tasklets

This section is discusses an alternative implementation for the RT-TTCN-3 code specification from Listing 6.8, Section 6.1.4, that presents the starting and stoping of test component at specified time points. Instead of creating a special stop task for stoping the targeted test component when its lifetime expires, we use timer tasklets. Each time a new timer is activated, there will be a tasklet, or timer handling routine associated with it, which is executed at every timer tick. Using this routine, we can trigger certain behaviors at precise points in time; such behaviors are, for example, the starting or stopping of a test component when the timer tick reaches a certain time value.

Listing 6.12 presents the body of the component task function. We can see that the component has a periodic behavior. All the activity for this component takes place in a loop, and at every iteration from that loop, the component gives up the processor for a fixed time period, using the library function `rt_task_wait_period()`, in Line 11.

```
/*** Component task structure */                        1
RT_TASK comp_task;                                      2
                                                        3
void comp_function(long comp_id) {                      4
    ...                                                 5
    while(active){                                      6
        /*** Test Component's behavior goes here */     7
        ...                                             8
        /*** Give up the processor for the period       9
         * of time that is indicated at activation */   10
        rt_task_wait_period();                          11
    }                                                   12
    return;                                             13
}                                                       14
```

Listing 6.12: Starting and Stopping a Test Component Using Timer Tasklets: Component Task Function

The definition of the timer handler routine is presented in Listing 6.13. This routine is executed every timer tick. Timer tick value is set by the `TICK_PERIOD` constant in Line 2 represents 50 microseconds. This means that every 50 microseconds, the routine `ph()` is executed. In this routine the current time is read, is then checked as to whether or not this value overpasses some thresholds. There are two such thresholds in our example.

Line 13 checks the first threshold, verifying whether or not the time point for starting the test component has been reached. If so, the parallel test component is activated with no delay, using the library method `rt_task_make_periodic_relative_ns()`. The

Table 6.3: RT-TTCN-3 To RTAI Mappings. Part IIIa.

RT-TTCN-3 Concepts	**RTAI C Code**

`NewPTC.start(BehaviorTestComponent())`
`at(now+100*microsec);`
	`#define DELAYSTART 100000`
	`rt_task_make_periodic_relative_ns(`
	       `&comp_task,`
	       `DELAYSTART,`
	       `0 ) ;`

`NewPTC.stop at(now+200*microsec);`	`#define DELAYSTOP 200000`
	`void comp_stop_function(long comp_stop_id){`
	     `...`
	     `rt_task_suspend(&comp_task );`
	     `rt_task_delete(&comp_task );`
	     `return;`
	`}`
	`...`
	`rt_task_make_periodic_relative_ns(`
	     `&comp_stop_task,`
	     `DELAYSTOP,`
	     `0);`
	`...`
	`int init_module(void){`
	     `...`
	     `rt_task_init(&comp_stop_task,`
	     `comp_stop_function,`
	     `comp_stop_id,`
	     `1024,`
	     `RT_SCHED_LOWEST_PRIORITY-3,`
	     `0,`
	     `0);`
	     `...`
	`}`
	`void cleanup_module(void){`
	     `...`
	     `rt_task_delete(&comp_stop_task );`
	     `...`
	`}`

period given as parameter of this function, `TICK_PERIOD`, represents the time period for which the newly activated test component can give up the processor. From now on, the test component is periodically executed in parallel with the timer handling routine.

The second threshold is checked in Line 24 and verifies whether or not there is the time for stoping the test component. If this is the case, then the parallel test component's execution is first, indefinitely suspended, then the parallel test component is deleted in Lines 28-29. Both `rt_task_suspend()` and `rt_task_delete()` operations are using the correspondent task structure as parameter in order to handle the right component task.

```
/*** 50 microseconds */                                              1
#define TICK_PERIOD 10000                                            2
...                                                                  3
/*** This routine is a timer handler routine and                     4
 * is automatically executed every timer period */                   5
void ph(unsigned long data){                                         6
        ...                                                          7
    /*** Here I get the current time in nanoseconds */                8
    crt_time = rt_get_time_ns();                                     9
    if(first_time){                                                  10
        /*** When the first time threshold is reached, the           11
         * corresponding PTC is activated */                         12
        if(crt_time >= start_time){                                  13
            ...                                                      14
            timestamp_compstart = rt_get_time_ns();                  15
            rt_task_make_periodic_relative_ns(&comp_task,            16
                                              0,                     17
                                              TICK_PERIOD);          18
        }                                                            19
    }                                                                20
    if(second_time){                                                 21
        /*** When the second time threshold is reached, the          22
         * corresponding PTC is killed */                            23
        if(crt_time >= stop_time){                                   24
            ...                                                      25
            timestamp_compstop = rt_get_time_ns();                   26
            comp_timespan = timestamp_compstop-timestamp_compstart;  27
            rt_task_suspend(&comp_task);                             28
            rt_task_delete(&comp_task);                              29
        }                                                            30
    }                                                                31
    return;                                                          32
}                                                                    33
```

Listing 6.13: Starting and Stopping a Test Component Using Timer Tasklets: Timer Handling Routine

Listing 6.14 presents the initializations that are needed for using timer tasklets. All the initializations are performed in the context of `init_module`. Times for starting and stopping the component are calculated in Lines 10-11. These timers are going to be used further in the context of the timer handler function for defining timer thresholds.

Depending on these thresholds, different actions might be taken. The timer and its associated tasklet are initialized in Lines 14-15, using the library functions defined in RTAI library `rtai_tasklets.h`, Line 1. `rt_insert_timer()`insert a timer in the list of timers to be processed.

Timers can be either periodic or oneshot. A periodic timer is reloaded at each expiration so that it executes with the assigned periodicity. A oneshot timer is fired just once and then removed from the timers list. Timers can be re-inserted or modified within their handler functions. The parameters of function `rt_insert_timer()`, as they can be observed in Lines 15-21, will be simply explained in the following section:

- First argument, `&pt`, is the pointer of the timer structure to be used to manage the timer at hand.
- Second argument, `0`, is the priority used to execute timer handlers when more than one timer has to be fired at the same time. It can be assigned any value such that: 0 < priority < `RT_LOWEST_PRIORITY`.
- Third argument, `rt_get_time()`, or firing time, is the time of the first timer expiration.
- Forth argument, `nano2count(TICK_PERIOD)`, is the period of a periodic timer. A periodic timer keeps calling its handler at

  ```
  firing_time + k*period k = 0, 1
  ```
- Fifth argument, `ph`, or the handler, is the timer function to be executed at each timer expiration.
- Sixth argument, `0`, or the data, is an unsigned long to be passed to the handler. Clearly, by an appropriate type casting, one can pass a pointer to whatever data structure and type is needed.
- Seventh argument, `0`, or pid, is an integer that marks a timer either as being a kernel or user space one. Despite its name you do not need to know the pid of the timer parent process in user space. Simple use 0 for kernel space and 1 for user space.

The component task is initialized in the usual manner, in Line 24. For deactivating the timer, library method `rt_remove_tasklet()` is used in Line 36.

The mappings from Tables 6.4 and 6.5 are alternative mappings, based on timers, to the ones from Table 6.3.

Table 6.4: RT-TTCN-3 To RTAI Mappings. Part IIIb.

RT-TTCN-3 Concepts	**RTAI C Code**

`NewPTC.start(BehaviorTestComponent())`
`at(now+100*microsec);`
	`#include <rtaitasklets.h>`
	`#define TICK PERIOD 10000`
	`...`
	`static struct rt_tasklet_struct pt;`
	`...`
	`void ph(unsigned long data){`
	`    ...`
	`    crt_time = rt_get_time_ns();`
	`    if(crt_time >= start_time){`
	`    ...`
	`    rt_task_make_periodic_relative_ns(`
	`       &comp_task,`
	`       0,`
	`       TICK_PERIOD);`
	`       }`
	`    ...`
	`    return;`
	`}`
	
	`int init_module(void){`
	`    ...`
	`    init_time = rt_get_time_ns();`
	`    start_time = init_time+2* TICK_PERIOD;`
	`    ...`
	`    rt_tasklet_use_fpu(&pt, 1);`
	`    rt_insert_timer(&pt,`
	`       0,`
	`       rt_get_time(),`
	`       nano2count(TICK_PERIOD),`
	`       ph,`
	`       0,`
	`       0) ;`
	`    ...`
	`    return 0;`
	`}`
	`void cleanup_module(void){`
	`    ...`
	`    rt_remove_tasklet(&pt);`
	`}`

TABLE 6.5: RT-TTCN-3 To RTAI Mappings. Part IIIc.

RT-TTCN-3 Concepts	**RTAI C Code**

`NewPTC.stop at(now+200*microsec);`	`#include <rtaitasklets.h>`
	`#define TICK PERIOD 10000`
	`...`
	`static struct rt_tasklet_struct pt;`
	`...`
	`void ph(unsigned long data){`
	`    ...`
	`    crt_time = rt_get_time_ns();`
	`    ...`
	`    if(crt_time >= stop_time){`
	`        ...`
	`        rt_task_suspend(&comp_task);`
	`        rt_task_delete(&comp_task);`
	`    }`
	`    ...`
	`    return;`
	`}`
	`int init_module(void){`
	`    ...`
	`    init_time = rt_get_time_ns();`
	`    ...`
	`    stop_time = init_time+4* TICK_PERIOD;`
	`    ...`
	`    rt_tasklet_use_fpu(&pt, 1);`
	`    rt_insert_timer(&pt,`
	`       0,`
	`       rt_get_time(),`
	`    nano2count(TICK_PERIOD),`
	`    ph,`
	`    0,`
	`    0) ;`
	`    ...`
	`    return 0;`
	`}`
	`void cleanup_module(void){`
	`    ...`
	`    rt_remove_tasklet(&pt);`
	`}`

```
#include <rtai_tasklets.h>
...
static struct rt_tasklet_struct pt;
...
int init_module(void){
    ...
    /*** Calculating the relative times for starting
     * and stopping the component */
    init_time = rt_get_time_ns();
    start_time = init_time + 2*TICK_PERIOD;
    stop_time = init_time + 4*TICK_PERIOD;

    /*** Activating the timer tasklets*/
    rt_tasklet_use_fpu(&pt, 1);
    rt_insert_timer(&pt,
                    0,
                    rt_get_time(),
                    nano2count(TICK_PERIOD),
                    ph,
                    0,
                    0);

    /*** Initializing a test component task */
    rt_task_init( &comp_task,
                  comp_function,
                  comp_id,
                  1024,
                  RT_SCHED_LOWEST_PRIORITY - 3,
                  0,
                  0);
    return 0;
}

void cleanup_module(void){
    ...
    rt_remove_tasklet(&pt);
    return;
}
```

LISTING 6.14: Starting and Stopping a Test Component Using Timer Tasklets: Init/-Cleanup Module

6.1.6 Receive Operation With Expiration Time

Listing 6.32 presents a RT-TTCN-3 code sample where a `receive` operation on a port is performed. It is interesting to notice how the waiting on the port is encapsulated in an `alt...break` construct in order to limit the expectation time – Listing 6.32, Lines 89-95. This means that expected event on the given port is waited for as long as the `break` condition is not violated. This means that the current time of the test system should not overpass the time limit indicated by the `break` operation. In the RT-TTCN-3 code snippet provided as an example, this time limit is one second after the start of the test case – see Line 95.

```
cont integer DEMO_MSG := 256;

type port DemoPortType message{
    inout integer DemoMsg;
}

type component MTCComponentType{
    const integer mtcID := 1;
    port DemoPortType receivePort;
    ...
    // other definitions
}

// this is the function containing the test behavior
testcase DemoTestCase() runs on MTCComponentType{
    var timespan ts := 1*sec;
    var float tf_receive := 0.0;
    ...
    alt{
        []receivePort.receive(DEMO_MSG) -> timestamp
        tf_receive {
                log("The expected message was received
                    at time ", seconds2timespan(tf_receive));
        }
    } break at(testcasestart + 1*sec){
        setverdict(fail);
        log("The message was not received in time");
    }
}
...
// the execution of the test case starts here
control{
    execute(DemoTestCase());
    ...
    // more test cases might follow
}
```

Listing 6.15: Time Restricted Receive Example with RT-TTCN-3

If the expected message has not been received yet and the time limit is violated, then the alternative behavior associated with the `break` operation is to be performed next. This consists, in most of the cases, of logging a message that indicates the failure, followed by setting a `fail` verdict.

The real-time implementation associated with the presented RT-TTCN-3 specification is listed in 6.17-6.19 code snippets.

For implementing the functionality of a timed receive operation we need four interacting real-time tasks: one for realizing the behavior of the test case, another for waiting on the port for the right message to be received, one for matching the received messages against the given pattern, and one for stopping the tasks for receiving and for matching if the time provided for `receive` operation has expired. Listing 6.16 presents these tasks by their task handler structure.

In order to simplify the explanations we are going to consider that the match operation is a trivial one and it is instantly performed on receiving the message. Therefore, we will exclude from discussion the match task and its interaction with the other treads, and concentrate mostly on showing the individual behavior and interactions between the three other threads.

```c
/*** Task structures that are used for implementing a
 * test case with a simple receive behavior with expiration
 * time */
RT_TASK tc_task;
RT_TASK receive_task;
RT_TASK match_task;
RT_TASK stop_task;
```

Listing 6.16: Receive Operation With Expiration Time:Used Task Structures

Listing 6.17 presents the task function associated with the receive task. This function contains a receive operation which is performed over the indicated port. In our case the port is implemented as a first in first out (FIFO) structure in Line 7. The type of FIFO that is used here is a real-time FIFO, provided by the RTAI. All the FIFO operations can be found within the `rtai_fifo.h` library. `rtf_get` operation tries to read a block of data from a real-time fifo previously created with a call to `rtf_create()`.

```c
                /*** Receive Operation With Expiration Time ***/      1
/***  This is the task that corresponds with the receive             2
 * operation */                                                      3
void receive_function(long receive_id) {                             4
        ...                                                          5
    /*** Here I wait for the message to arrive */                    6
    while(!(retval = rtf_get(FIFO, &msg, sizeof(int))));             7
    rt_task_suspend(&stop_task);                                     8
    rt_task_delete(&stop_task);                                      9
    ...                                                              10
    return;                                                          11
}                                                                    12
```

Listing 6.17: Receive Operation With Expiration Time:Receive Task Function

Listing 6.18 presents the function associated with the stop task. This task is responsible for stoping the receive task from listening on the port after the given time for that operation expires. The stopping of the receive task implies suspending and then, deleting it. On the other hand, if the expected message was received in time, it can also be noted from Listing 6.17, that the receive task suspends and deletes the stop task that is associated with it.

```
/*** This task is activated when the given time for the receive
 * operation expires */
void stop_function(long stop_id) {
    /*** Suspend the receive task */
    rt_task_suspend(&receive_task);
    /*** Delete the receive task */
    rt_task_delete(&receive_task);
    return;
}
```

Listing 6.18: Receive Operation With Expiration Time:Stop Task Function

Listing 6.19 makes the actual connection between the receive and the stop tasks. In the body of the test case task function that is presented here, the receive task is activated without delay, while the stop task is set to become active one second after from the beginning of the test case expiration. The initializations/deinitialization for these tasks are performed in `init_module/cleanup_module` and they are similar to the ones presented in previous sections. Therefore, they will not be emphasized here.

```
/*** Task function that implements the behavior of the test case */
void testcase_function(long tc_id) {
    RTIME delay = 1000000000;     /*** 1 second */

    /*** The receive task is activated */
    rt_task_resume(&receive_task);

    /*** If the maximum receive period expires, the receive
     * task will be killed by the stop task; 1 second
     * is the receive interval here. */
    rt_task_make_periodic_relative_ns(&stop_task,
                                      now + delay,
                                      0);
    return;
}
```

Listing 6.19: Receive Operation With Expiration Time:Testcase Task Function

The specification to code mappings presented in this section, are summarized in Table 6.6. There is only one analyzed operation, namely a `receive` that is time-bounded by an `alt..break` statement.

6.1.7 Send Operation

This section exemplifies the implementation for a `send` operation that should be performed at a predefined moment in time. The section is structured in the same manner as the previous section. Listing 6.20 presents the RT-TTCN-3 specification which contains a send operation on a port. The operation should be effectuated at one second after

Table 6.6: RT-TTCN-3 To RTAI Mappings. Part IV.

RT-TTCN-3 Concepts	**RTAI C Code**

`alt{`
`    [] receivePort.receive(DEMO_MSG)`
`       {`
`       ...`
`       }`
`}break at(testcasestart + 1*sec)`
`{`
`    setverdict(fail);`
`    log("The message was not`
`         received in time");`
`}`
	`void receive_function(long receive_id){`
	`     ...`
	`     // receive operation here`
	`     while(!(rtf_get(FIFO,`
	`                    &msg,`
	`                    sizeof(int))));`
	`     rt_task_suspend(&stop_task);`
	`     rt_task_delete(&stop_task);`
	`     ...`
	`     return;`
	`}`
	
	`void stop_function(long stop_id){`
	`     rt_printk("The message was not`
	`         received in time");`
	`     rt_task_suspend(&receive_task);`
	`     rt_task_delete(&receive_task);`
	`}`
	
	`void testcase_function(long tc_id){`
	`     RTIME delay = 1000000000;`
	`     rt_task_resume(&receive_task);`
	`     rt_task_make_periodic_relative_ns(`
	`        &stop_task,`
	`        now + delay,`
	`        0);`
	`     return;`
	`}`

the start of the test case, as indicated in Line 123. Representative excerpts from the implementation of this specification are provided in Listings 6.21-6.25.

Listing 6.21 presents the task structures that are needed for implementing the behavior of a time triggered send operation: there is a task associated with the testcase and another task that is associated with the send operation. The task function of the send operation is described further, in Listing 6.22. The operation of sending a message is realized, in our example, over a FIFO channel in Line 12. The `rtf_put_if()` method that is used here, tries to write a block of data to a real-time fifo previously created with `rtf_create()`.

```
cont integer DEMO_MSG := 256;

type port DemoPortType message{
    inout integer DemoMsg;
}

type component MTCComponentType{
    const integer mtcID := 1;
    port DemoPortType sendPort;
    ...
    // other definitions
}

// this is the function containing the test behavior
testcase DemoTestCase() runs on MTCComponentType{
    ...
    sendPort.send(DEMO_MSG) at (testcasestart + 1*sec);
    ...
}
```

Listing 6.20: Send At Example with RT-TTCN-3

```
/*** Task structure that are used for implementing a
 * test case with a simple send behavior at a given
 * time */
RT_TASK tc_task;
RT_TASK send_at_task;
```

Listing 6.21: Send Operation: Task Structures

```
/*** Send at a Given Time Point. (FIFO Communication Channel) ***/
...
#include <rtai_fifos.h>
...
#define FIFO 10
bool active = true;
...
/*** Task function that implements the send operation */
void send_at_function(long send_at_id) {
    while(active){
        /*** Send operation goes here */
        rtf_put_if(FIFO, &i, sizeof(int));
        rt_task_wait_period();
    }
    return;
}
```

Listing 6.22: Send at Operation: Send At Task Function

For recording the timestamp of a send operation, the time of the system is interrogated and saved, just before the actual sending is performed. This is done in a simple manner, using the `rt_get_time_ns()` function, as presented in Listing 6.23.

```
...
/*** Recording the time goes here */
RTIME send_timestamp = rt_get_time_ns();
/*** Send operation goes here */
rtf_put_if(FIFO, &i, sizeof(int));
...
```

Listing 6.23: Send Operation with Timestamp

Listing 6.24 presents the task function implementing the behavior of the test case. In Line 6 the send task is activated and set to start one second after the beginning of the testcase.

```
 1  void testcase_function(long tc_id) {
 2      RTIME start_delay = 1000000000;  /*** 1 second */
 3      /*** Activates send at operation and the first
 4       * activation time has a relative delay of 1
 5       * second */
 6      rt_task_make_periodic_relative_ns(&send_at_task,
 7                                        start_delay,
 8                                        start_delay);
 9      ...
10      return;
11  }
```

Listing 6.24: Send at Operation: Testcase Task Function

The `send` operation was implemented in our example using a FIFO communication channel as a commiunication port. Listing 6.25 presents the operations for initiaizing/uninitializing such a channel. `rtf_create()` creates a real-time fifo (RT-FIFO) of initial size `MAX_SIZE` and assigns it the identifier fifo. It must be used only in kernel space. The RT-FIFO is a character based mechanism used to communicate among real-time tasks and ordinary Linux processes. The `rtf_*` functions are used by the real-time tasks.

```
/*** Initialize/destroy the communication channel (port);
 * In this case, it is a FIFO. */
rtf_create(FIFO, MAX_SIZE);
rtf_destroy(FIFO);
```

Listing 6.25: Send at Operation: FIFO Operations

The section ends with the table summarizing the mapping between abstract specification and its implementation. Table 6.7 is a follow up of the tables presented in previous sections.

TABLE 6.7: RT-TTCN-3 To RTAI Mappings. Part V.

RT-TTCN-3 Concepts	**RTAI C Code**

`sendPort.send(DEMO_MSG)`
`         at (testcasestart + 1*sec);`
	`void send_at_function(long send_at_id){`
	`     while(active){`
	`         rtf_put_if(FIFO,`
	`             &i,`
	`             sizeof(int));`
	`         rt_task_wait_period();`
	`     }`
	`     return;`
	`}`
	`void testcase_function(long tc_id){`
	`     RTIME start_delay = 1000000000;`
	`     ...`
	`     rt_task_make_periodic_relative_ns(`
	`         &send_at_task,`
	`         start_delay,`
	`         start_delay);`
	`     ...`
	`     return;`
	`}`
	`int init_module(void) {`
	`     ...`
	`     rt_task_init(&send_at_task,`
	`         send_at_function,`
	`         send_at_id,`
	`         1024,`
	`         RT_SCHED_LOWEST_PRIORITY-3,`
	`         0,`
	`         0);`
	`     ...`
	`     return 0;`
	`}`

6.1.8 Real-time Sockets. Receive Server.

This section shows how real-time communication on sockets can be implemented. Some relevant examples using real-time sockets, listed in 6.26-6.29, illustrate the functionality of a receive server. The listings from the next section, Listings 6.30-6.31 are completing the image, by illustrating a complementary functionality to the receive server, that of a `send` client. These two applications are exchanging messages between eachother in real-time. In this manner, both being real-time applications, one can be regarded as *SUT*, and the other one as *TS*, interchangeable.

All the presented examples that use real-time sockets to communicate, are based on the RTnet [119] extension for RTAI. RTnet is an open source, hard real-time network protocol stack for Xenomai and RTAI. It makes use of standard Ethernet hardware and supports several popular NIC chip sets, including Gigabit Ethernet. RTnet implements UDP/IP's, TCP/IP's basic features, and provides a POSIX socket API to real-time user space processes and kernel modules [119]. For the presented examples, the version of RTnet that was used is version 0.9.12, the last one available at the date of this implementation.

In the following section we are going to discuss the code for the server side. The correspondent RT-TTCN-3 test specification implemented by this code can be the one presented previously in Listing 6.32. If we change the implementation of the port, the test specification remains the same. The port in this case, is a real-time socket instead of a FIFO. All the excerpts of code that are shown in here and refer to the socket implementation, should belong, in TTCN-3 terminology, to the test adaptor.

Listing 6.26 presents the libraries and declarations of constants and variables that are necessary to implement the server. The library that contains methods for handling real-time sockets is called `rtnet.h`. In Line 9 we defined the port on which the server listens for new incoming messages. The socket structure is defined in Line 12. The task handlers that are needed for constituting the server are declared in Lines 16-19. There is one `receive` task, which waits for the message to come, one match task which compares the arrived messages against a predefined pattern, the stop task that kills the receive and match tasks when the time expires and a test case task, containing the behavior of the test case. They are fundamentally the same tasks presented in Section 6.1.6, and as in the aforementioned section, the same pattern for task interactions is used.

Listing 6.27 presents the task function definitions for the stop task and for the match task. Stop task is responsible with killing the `receive` and associated match tasks when predefined time expires. The functionality of the stop task, line 4, is the same one as presented in previous sections. The match task communicates with the `receive` task through a real-time FIFO, lines 16-17. Every time a new message is received, the match task is awoken and starts the matching. Matching means comparing the newly received message against a predefined template or set of templates. If the match succeeds, then it means that the right message has been received by the *TS*, and that the *TS* can go

```
...
#include <linux/net.h>
#include <linux/socket.h>
#include <linux/in.h>
#include <rtai_mbx.h>
#include </usr/local/rtnet/include/rtnet.h>
...
/*** This is the port on which the server is listening to */
#define PORT 8888
...
/*** Local socket */
static struct sockaddr_in local_addr;
static int sockfd = 0;
static char msg[4000];
/*** The task structures that are used here. */
RT_TASK receive_task;
RT_TASK stop_task;
RT_TASK match_task;
RT_TASK tc_task;
SEM FIFO_sem;
...
```

Listing 6.26: Real-time Sockets. Receive Server. Declaration Part.

forward with other testings. Once the match is validated, the match task is the one that kills the stop task this time, lines 20-21. In this context, the stop task is the one which becomes useless, since the right message has been received in time. Also, the receive task should also be killed after the match is validated, as it does not make sense to keep on waiting for a message that has already arrived, 22-23.

Further on, in Listing 6.28, the actual receive task function is presented, this represents the kern functionality of the receive server. It is important to observe here, how the RTnet library methods are used for binding the real-time socket with the given local address and port, line 6, and for receiving the incoming messages on that port, line 11. `rt_dev_bind()` is the method for binding. It receives, as parameters, the handler of the socket, `sockfd`, and the socket structure which should have been previously initialized and created in the `module_init` section, as shown in Listing 6.29, Lines 4-12.

The code snippet 6.29 presents how the socket is initialized, Lines 4-11, created, Line 12, and in the end, closed, Line 19. The routines for dealing with the real-time sockets are similar to the Berkeley sockets. The difference is that real-time sockets are implemented with real-time system calls and priorities.

6.1.9 Real-time Sockets. Send Client.

This section discusses the implementation of a real-time client that sends messages to a real-time receive server. The focus of the client operation is to realize the send operation at fixed, predefined times. The communication channel is implemented as a real-time

```
  /*** function associated with the stop task,
   * responsible to upper bound the receive
   * operation */
void stop_function(long stop_id) {
    rt_task_suspend(&receive_task);
    rt_task_delete(&receive_task);
    return;
}

void match_function(long match_id){
    bool match = false;
    /*** Here the matching of the received message is realized */
    /* if the message matches, the associated receive task
     * is suspended; else, no action is taken */
    while(true) {
        rt_sem_wait(&FIFO_sem);
        rtf_get(FIFO, &msg, sizeof(msg));
        /*** Realizes the match against the template here */
        if(match){
            rt_task_suspend(&stop_task );
            rt_task_delete(&stop_task );
            rt_task_suspend(&receive_task );
            rt_task_delete(&receive_task );
            break;
        }
    }
    return;
}
```

LISTING 6.27: Real-time Sockets. Stop And Match Task Functions.

```
/***  This is the task that corresponds with the receive
 * operation */
void receive_function(long receive_id) {
    ...
    /*** Bind socket to local address */
    rt_dev_bind(sockfd,
        (struct sockaddr *) &local_addr,
        sizeof(struct sockaddr_in));
    ...
    /*** Block until packet is received */
    rt_dev_recv(sockfd, msg, sizeof(msg), 0);

    /*** Message was received */
    rt_printk("The message=%s was received, at time = %lld\n",
              msg, rt_get_time_ns());
    /*** Put the message on the FIFO and signal the task which
     * executes the matching of the message */
    rtf_put_if(FIFO, &msg, sizeof(msg));
    rt_sem_signal(&FIFO_sem);
    ...
    return;
}
```

LISTING 6.28: Real-time Sockets. Receive Task Function.

```
int init_module(void) {
    ...
        /*** Initialize the socket */
    memset(&local_addr,
           0,
           sizeof(struct sockaddr_in));
    local_addr.sin_family = AF_INET;
    local_addr.sin_addr.s_addr = INADDR_ANY;
    local_addr.sin_port = htons(PORT);

        /*** Create a new socket */
    sockfd = rt_dev_socket(AF_INET, SOCK_DGRAM, 0);
    ...
    return 0;
}
void cleanup_module(void) {
    ...
    /*** Close the socket */
    rt_dev_close(sockfd);
    ...
    return;
}
```

Listing 6.29: Real-time Sockets. Initialize the sockets.

socket. The code examples in 6.30-6.31 illustrate how to create a client connection on a real-time socket and emphasize the methods that are used for that.

Listing 6.30 shows the implementation of the `send` task function, which is responsible for sending the message after it starts. Before the actual send operation is performed, the binding to the local socket should be first realized, followed by establishing a connection with the socket from the server side. As it results, this time there are two sockets being handled: the local socket, used when the message leaves the system, and the remote socket of the server, which represents the destination address to which the message is sent. The two corresponding socket structures are declared in Lines 3-4. The binding to the local socket is realized in Line 9 and the connection to the remote server is realized in Line 15. Afterwards, the message can be sent, as shown in Line 20.

Listing 6.31 performs the initializations that regard sockets in Lines 4-13. This time two socket structures are used instead of one. The local socket is created in Line 15, and closed in Line 21.

6.2 FreeRTOS Based Real-time Testing Framework. Auto Car Door Case Study.

While the first part of this chapter has discussed different mappings between newly introduced real-time concepts for TTCN-3 and RTAI, here we will challenge the capabilities

```
...
/*** Socket structures for local and remote address */
static struct sockaddr_in local_addr;
static struct sockaddr_in server_addr;

void send_function(long send_id) {
    /*** Bind socket to local address specified as
     * parameter */
    rt_dev_bind(sockfd,
        (struct sockaddr *)&local_addr,
        sizeof(struct sockaddr_in));
    ...
    /*** Specify destination address for socket;
     * needed for rt_socket_send */
    rt_dev_connect(sockfd,
        (struct sockaddr *)&server_addr,
        sizeof(struct sockaddr_in));
    ...
    /*** Send a message */
    rt_dev_send(sockfd, msg, sizeof(msg), 0);
    ...
    return;
}
```

Listing 6.30: Real-time Sockets. Send Client.

```
int init_module(void) {
...
/*** Initialize the sockets */
memset(&local_addr, 0, sizeof(struct sockaddr_in));
memset(&server_addr, 0, sizeof(struct sockaddr_in));
/*** Local socket */
local_addr.sin_family = AF_INET;
local_addr.sin_addr.s_addr = INADDR_ANY;
local_addr.sin_port = htons(PORT);
/*** Remote socket*/
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = rt_inet_aton(SRV_IP_ADDR);
server_addr.sin_port = htons(SRV_PORT);
/*** Create new socket */
rt_dev_socket(AF_INET, SOCK_DGRAM, 0);
...
return 0;
}
void cleanup_module(void) {
    /*** Close the socket */
    rt_dev_close(sockfd);
    ...
    return;
}
```

Listing 6.31: Real-time Sockets. Init/Cleanup Module.

of the second chosen real-time operating system, the FreeRTOS platform. In the previous section we used small examples of possible test system implementation segments to prove the usage of the RT-TTCN-3 concepts and the RTAI mechanism that are used at their realization.

In this section we are going to build a real-time test case example, implemented within FreeRTOS. This is going to be used for testing a concrete real-time automotive application, consisting of a control system embedded into a car's door. The embedded system controls simple movements of the car door's parts: e.g. lifting up or down the door's window, or turning on and off the flash light installed on the car's door. The simple movements might be combined in order to constitute more complex automatic behaviors, e.g. the crash state: the window goes automatically down, while the flash light starts blinking on-off, on-off... The communication between the test system and the *SUT* represented by the car door controller is going to be realized through a serial cable, requiring a COMM implementation for the port on which the message is sent or received, respectively.

6.2.1 Auto Car Door Case Study

The actual mapping of the basic and newly introduced concepts of the TTCN-3 language to a RTOS platform is explained by a simple test case for an example taken from the automotive domain. The chosen example consist of an embedded system on a MC9S12NE64 demo board [120] attached to a car door. The system controls various basic units of the door, for example, the window lifter, flashing light indicator, electrical central locking system. The system changes its internal state when receiving signals via RS232 serial interface. These signals are in fact, certain control strings with the role of triggering a special basic functionality of the door, for example, driving up the window, turning on the flash indicator and so forth. When receiving such a string, the system enters a different state, executes the function associated with that state, and sends back a response string to the serial interface. The basic functions of the door could be combined in order to form safety applications such as when an accident occurs, the window should be driven down, the flash indicator should blink and the door should be automatically unlocked. Therefore, it is important to assure that the basic functionality is happening in real-time. One can put several real-time constraints on this, such as: "The window should be driven down within one millisecond; the flash signal should be turned on within one millisecond; it should remain on for 5 $\times$ milliseconds with a tolerance of $\pm$ 1 millisecond, then it should turn off."

Figure 6.1 presents the test setup. On the left side we have the embedded system connected to the door, and on the right there is the *TS* consisting of a PC with FreeRTOS installed. The PC port of FreeRTOS that we are utilizing runs on an integrated environment from the WATCOM open source project [121] (the distribution is for Windows). The PC is connected to the board through a serial cable. The real-time requirement that one wishes to test can be formulated in this manner: "The flash signal should be turned on within one millisecond". For testing this requirement only, the signal is sent several times with the instruction of repeated flashing. After sending one signal, when the flash is on, it is assumed that it automatically turns off after a fixed period of time. After expiring this period of time, one can send another signal for turning it on again. This example epitomizes the newly introduced concepts.

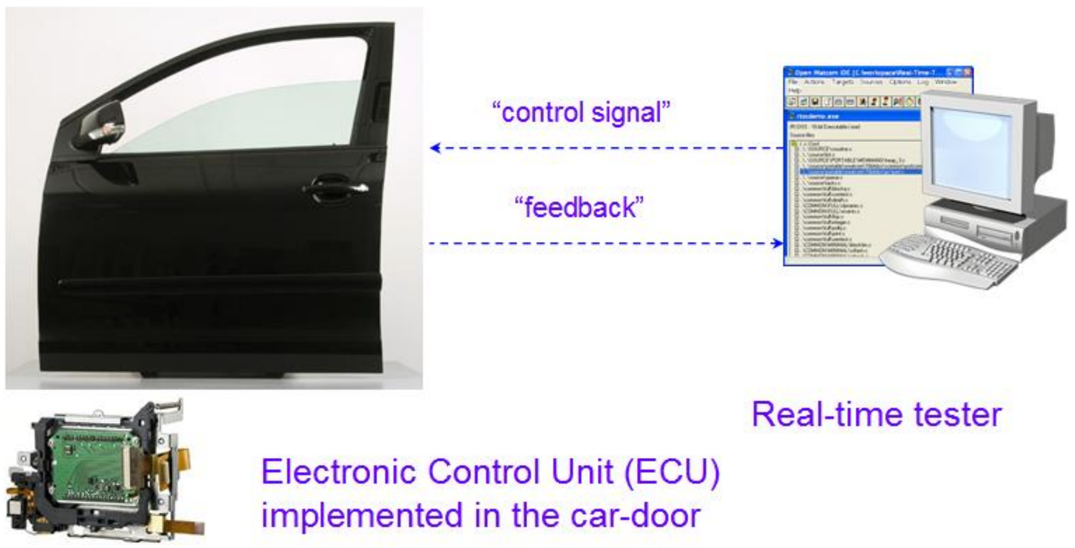


FIGURE 6.1: Auto Car Door Demo Setup And Configuration [3]

The test specification written in RT-TTCN-3 is listed in 6.32. The presented test case tests the following real-time behavior of the *SUT*: after the *SUT* is initialized, by sending the `INIT` signal towards it, it waits for a `MIN_INTERVAL` period in order for the initialization to take place, then the `Time_Blinker_ON` stimulus is sent; this stimulus should trigger the blinking of the flash light on the *SUT*, with a fixed time period between the "off" and "on" modes of the *SUT*; the fixed time period is required to be of approximate 5 milliseconds, with a maximum error of ± 1 millisecond, indicated by the `MIN_INTERVAL`. The maximum time interval on which the port will listen for message receival is 0.2 seconds, the value set for `MAX_INTERVAL`.

The input/output signals for communicating with the *SUT* are the ones defined in Lines 127-132. Further on, the timings are defined in the following lines, 133-135. The test case starts with sending the `INIT` signal for initializing the *SUT*, then it waits for `MIN_INTERVAL` for the initialization to take place. In Line 143 the current time is read, in order to be kept as an anchor point for further time measurements. The `Time_Blinker_ON` signal is sent to the *SUT* to trigger the blinking behavior on its side.

```
...
template charstring INIT := "*";
template charstring  Time_Blinker_ON := "4";
template charstring  Time_Blinker_OFF := "5";
template charstring RESET := "reset";
template charstring ON := "4Blink LED On";
template charstring OFF := "5Blink LED Off";
const timespan OFF_ON_INTERVAL := 5*millisec;
const timespan MAX_INTERVAL := 0.2*sec;
const timespan MIN_INTERVAL := 1*millisec;

// this is the function containing the test behavior
testcase CarDoorDemoTestCase() runs on MTCComponentType{
    var float starttime, ON_timestamp, OFF_timestamp,
              DIFF_timestamp;
    commSerialPort.send(INIT);
    wait(MIN_INTERVAL);
    starttime := now;
    commSerialPort.send(Time_Blinker_ON);

    while(true){
        alt{
            [] commSerialPort.receive(ON) -> timestamp
                ON_timestamp{
                log("The ON message was received
                    at time ", seconds2timespan(ON_timestamp));
                continue;
            }
            [] commSerialPort.receive(OFF) -> timestamp
                OFF_timestamp{
                log("The OFF message was received
                    at time ", seconds2timespan(ON_timestamp));
                DIFF_timestamp:= OFF_timestamp-ON_timestamp;
                if(OFF_ON_INTERVAL-MIN_INTERVAL <=
                    DIFF_timestamp <=
                    ON_OFF_INTERVAL+MIN_INTERVAL)
                    continue;
                            else{
                    setverdict(fail);
                    log("wrong timing");
                    break;
                }
            }
        } break at(starttime + MAX_INTERVAL){
            commSerialPort.send(Time_Blinker_OFF);
                        wait(MIN_INTERVAL);
                        commport.send(RESET);
            setverdict(fail);
            break;
        }
    }
}
...
```

Listing 6.32: Auto Car Door Demo Example with RT-TTCN-3

Line 148 presents a receive operation with timestamp that is expecting messages of "on"-type, while in Line 154 the timestamped receive is waiting for messages of "off"-type. Every time "on"-"off" messages alternate, the interval between "off" and "on" is calculated and compared with the time expectation; see Line 158. If time interval conforms to the one from the specification, the testing may continue, otherwise an error message is displayed, together with a fail verdict.

```
 ...
#include "FreeRTOS.h"
#include "task.h"
#include "serial.h"
...
/*** Priority definitions for the tasks in the demo
 * application. */
#define mainMTC_TASK_PRIORITY    (tskIDLE_PRIORITY + 1)
/*** Stack size */
#define STACK_SIZE  ((unsigned portSHORT)1024)
/*** Constant definition used to turn on/off the preemptive
 * scheduler. */
static const portSHORT sUsingPreemption = configUSE_PREEMPTION;
/*** Handle to the com port used by both tasks. */

static xComPortHandle xPort;
static const portLONG xBlockTime = 10000000000;
static const eCOMPort ePort = serCOM1;
static const eBaud eBaudRate = ser115200;
/*** Predefined message codes for communicating with the SUT */
portCHAR codes[][2] = {"*", "4"};
portCHAR responses[][30] = {"Initialisierung durchlaufen",
                            "4Blink LED On"};
...
/*** Prototypes for the Main Component Task and tSend and tExp
 * tasks*/
static void vMTC( void *pvParameters );
void v_tSend( void *pvParameters );
void v_pReceive( void *pvParameters );
void v_tExp(void *pvParameters);
/*** Method that realizes the matching between the received message
 * and the expected template */
portBASE_TYPE match(portCHAR* set1, portCHAR* set2);

```

Listing 6.33: Declaration and Definition Section.

In the following Listings, from 6.33 to 6.39, the code examples that illustrates how the above specification is implemented in C for the FreeRTOS environment are presented. The interesting aspects unveiled by those examples are the look and feel of the FreeRTOS specification and the handling of COMM serial port as a communication port between the real-time *TS* and the real-time *SUT*.

Listing 6.33 presents the declaration section of the real-time *TS* implementation. It shows the important libraries that are used: `FreeRTOS.h` is the library providing kern methods of FreeRTOS, as for example, the method that starts the real-time scheduler; `task.h` library provides the methods related to task creation and manipulation; `serial.h` library contains methods for working with the serial communication channel. Line 13 shows how the scheduler is set to allow preemption, by setting the `sUsingPreemption` constant. Between Lines 16-19 the variables used at configuring the serial port are comprised. The strings and templates for communicating with the *SUT* are defined in 21-23. It can be seen from Lines 27-34, which are the interacting tasks that are realizing the behavior of the previously presented RT-TTCN-3 specification.

```
 1  /*** main function is where the MTC is created */
 2  portSHORT main( void ) {
 3      xTaskHandle mtcHandle;
 4      /*** Create the co-routines that communicate with the
 5       * tick hook. */
 6      vStartHookCoRoutines();
 7      /* Create the "MTC" task */
 8      xTaskCreate( vMTC,
 9                   "TMTC",
10                   STACK_SIZE,
11                   &mtcHandle,
12                   mainMTC_TASK_PRIORITY,
13                   &mtcHandle );
14      /*** Set the scheduler running.  This function will not
15       * return, unless a task calls 'vTaskEndScheduler' . */
16      vTaskStartScheduler();
17      return 1;
18  }
```

Listing 6.34: Create Main Test Component

Further on, Listing 6.34 presents the main function, which contains the main initialization of the real-time *TS* application. In Line 8 the task corresponding to the MTC is created using the `xTaskCreate()` routine. This creates a new task and adds it to the list of tasks that are ready to run. The opposite to this operation is `xTaskDelete()`, which removes a task from the RTOS real-time kernel's management. It is interesting to observe that, in Line 16, the real-time scheduler is started using the `vTaskStartScheduler()`. This starts the real-time kernel tick processing. After this calling, the kernel has control over which tasks are executed and when. The idle task is created automatically when `vTaskStartScheduler()` is called. If `vTaskStartScheduler()` is successful the function will not return until an executing task calls `vTaskEndScheduler()`. The function might fail and return immediately if there is insufficient RAM available for the idle task to be created.

```
 1  /*** The behavior of the MTC is described here */
 2  static void vMTC(void *pvParameters)
 3  {
 4          /*** Initializations */
 5          xTaskHandle selfHandle = *(xTaskHandle *)pvParameters;
 6          xTaskHandle tSend_Handle;
 7          xTaskHandle tRcv_Handle;
 8          ...
 9          /*** Initialize the COM port. */
10          xPort = xSerialPortInit( ePort,
11                                   eBaudRate,
12                                   serNO_PARITY,
13                                   serBITS_8,
14                                   serSTOP_1,
15                                   uxBufferLength );
16          /*** Create the tSend and tRcv tasks */
17           xTaskCreate( v_tSend,
18                        ”TSEND” ,
19                        STACK_SIZE,
20                        &params,
21                        mainMTC_TASK_PRIORITY,
22                        &tSend_Handle );
23           xTaskCreate( v_pReceive,
24                        ”PRCV” ,
25                        STACK_SIZE,
26                        &params,
27                        mainMTC_TASK_PRIORITY,
28                        &tRcv_Handle );
29          ...
30          /*** Suspend self */
31           vTaskSuspend( NULL );
32          return;
33  }
```

Listing 6.35: Main Test Component's Task Function.

Listing 6.35 implements the task function associated with MTC. The main actions that the MTC performs is that of initializing the serial port, in Line 10, and creating the tasks for send and receive, in Lines 17-23. After the MTC task executes these operations, it suspends itself, in Line 31, using the `vTaskSuspend()` function. This function suspends any task. When suspended, a task will never get any microcontroller processing time, no matter what its priority.

Listing 6.36 presents the task function that sends the messages on the serial port. The sending of messages is realized with the `vSerialPutString()` function. After the message activating the blinking is sent to the *SUT*, a timer task is created, on Line 25. This timer task is responsible for finishing the receive task if the maximum amount of time, for which a message is expected, overpasses. The implementation for the timer task is presented further in Listing 6.37.

```c
void v_tSend(void *pvParameters){
    /*** Initializations */
    t_params params = *(t_params *) pvParameters;
    ...
    for( ;; ){
        /* Suspend the current task for a given time */
        vTaskDelay(tSendConst);
        if(codeId == 0){
            /*** Send message on the serial port */
            vSerialPutString( xPort, codes[codeId],
                                strlen(codes[codeId]));
            codeId++;
        }else if(codeId == 1 && i < NR_OF_TIMES) {
            ...
            /*** Send the string to the serial port and activate
             * timer */
            time1[i] = xTaskGetTickCount();
            vSerialPutString( xPort, codes[codeId],
                                strlen(codes[codeId]));
            printf("Code '%s' sent at %ld\n", codes[codeId],
                        time1[i]);
            /*** Create the tExp task in order to wait for the
             * response for this data set */
            memset(tExpName, 0, sizeof(tExpName));
            xTaskCreate( v_tExp,
                         tExpName,
                         STACK_SIZE,
                         &i,
                         mainMTC_TASK_PRIORITY,
                         &tExpHandle[i] );
            i++;
        }
    }
}
```

Listing 6.36: Send Task. COM Serial Port.

```c
void v_tExp(void *pvParameters){
    /*** Initializations */
    ...
    setId = *(portSHORT *) pvParameters;
    ...
    sprintf(verdict,"SET VERDICT FAIL\r\n");
    for(;;){
        /*** Here is indicated the time after which
         * the timer function becomes active */
        vTaskDelay(tExpConst);
        vDisplayMessage(timeout);
        vDisplayMessage(verdict);
        /*** All test system is stopped */
        vTaskEndScheduler();
    }
}
```

Listing 6.37: Timer Expiration Task

The main behavior of timer task function is that of waiting for the maximum expiration time to pass, which is associated with the `receive` operation. If the time expires and the timer task is still alive, then it will end the execution of the test case, by invoking the `vTaskEndScheduler()` function, and it will generate a fail verdict. If the message will be received in time, then the receive task is going to kill the timer task before this would have the chance to become active, see Listing 6.38, Line 27.

```
1  void v_pReceive(void *pvParameters){
2      /*** Initializations */
3      t_params params = *(t_params *) pvParameters;
4      for( ;; ){
5      /*** Receive a message from the interrupt routine associated
6       * with the COM port. If a message is not yet available the
7       * call will block the task. */
8      while(1) {
9          xGotChar = xSerialGetChar( xPort,
10                                    &cRxedChar,
11                                    xBlockTime );
12         if( xGotChar == pdTRUE ) {
13             if(resLen < strlen(responses[resId])){
14                 sprintf(responseStr+strlen(responseStr),
15                     "%c",cRxedChar);
16                 resLen++;
17                 if(resLen == strlen(responses[resId])){
18                     /*** Validate the response */
19                     if(match(responseStr,responses[resId])){
20                         /*** Recording the timestamp of the message
21                          * receival */
22                         time2[i] = xTaskGetTickCount();
23                         printf("Response |%s| receieved at %ld\n",
24                             responseStr, time2[i]);
25                         /*** Delete the timer associated with the
26                          * response */
27                         vTaskDelete(tExpHandle[i]);
28                         /*** Set verdict to PASS */
29                         sprintf(verdict,"SET VERDICT PASS\r\n");
30                         ...
31                         if(finished == NR_OF_TIMES){
32                             vTaskEndScheduler();
33                             break;
34                         }
35                     }
36                 }
37             }
38         }else{
39             vDisplayMessage(pcTaskTimeoutMsg);
40         }
41     }
42     }
43 }
```

Listing 6.38: Receive Task. COM Serial Port.

Listing 6.38 presents the task function responsible for intercepting the messages from the serial port. The interception is realized in Line 9.

```
portBASE_TYPE match(portCHAR* code1, portCHAR* code2){
    portBASE_TYPE matched = pdTRUE;
    if(strcmp(code1,code2) != 0)
        matched = pdFALSE;
    return matched;
}
```

LISTING 6.39: Match Operation.

The match operation for comparing incoming messages with the expected templates is illustrated in Listing 6.39.

As proven by the code excerpts presented in this chapter, the mappings between real-time TTCN-3 concepts and real-time operating system instructions result in modular, compact blocks of instructions for both chosen platforms. The mappings are relatively straight forward and easy to grasp. Different combinations of those blocks can be used to cover the entire space of possibilities for the behaviour of the real-time test system.

The next chapter presents the benchmark of these implementations, by a means of WCET for Linux with RTAI and by means of test results of a real test case study, for the FreeRTOS.

6.3 Summary

This chapter was split in two main sections. Each section presented the realization of the real-time TTCN-3 *TS* on a different real-time operating system platform. The first section presented the mappings of the individual language extensions on RTAI Linux specific mechanisms, realized in C. The Real-time TTCN-3 *TS* is, in this context, a kernel module with hard real-time abilities. RTAI Linux is a complex real-time operating system, that is general purpose, in the sense that it offers scheduling mechanisms appropriate for an extensive set of tasks, with complex inter-task communication mechanisms, allowing the build of a wide range of real-time application. In contrast to this, FreeRTOS kernel is small and minimal and offers only a basic set of services. The second part of this chapter was dedicated to a case study taken from the automotive domain. It presents the testing of an ECU, controlling the functionality of an automatic car door. This time, the *TS* required for testing this special purpose controller, was built on top of FreeRTOS operating system. FreeRTOS'es simplicity and conciseness means it is recommended as a better option for building upon the testing of such a small embedded application. We found it particularly challenging to define the mappings of the real-time concepts of extended TTCN-3 to the minimal set of services provided by FreeRTOS platform.

Chapter 7

Results And Discussion

"However beautiful the strategy, you should occasionally look at the results."
Sir Winston Churchill

The first part of this chapter provides a benchmark of the *RTTS* implemented using the mappings presented in Chapter 6. The results and the evaluations of the implemented *RTTS* will be presented and discussed, with the highlights set upon the newly introduced concepts and operations. The benchmark approach adopted here is inspired from the *Individual Language Feature Benchmarks* strategy developed for Ada [9], [8].

Each of the different subsections of this chapter is dedicated to one real-time construct such as: `wait`, `start at`, `stop at`, `send at`, or `receive`...`break at`. Each construct has some precise time requirements associated with it. Through specific *TS* configuration and implementation, each of the individual aforementioned operations is isolated and evaluated at runtime. Maximum latencies and worst case execution times are calculated accordingly, based upon the experimental results for each of the presented test configurations.

The second part of this chapter evaluates the results for the AutoDoo Case Study.

7.1 Benchmark Of A Test System Implemented On Linux With RTAI Real-time Operating System

"The worst-case execution time (WCET) of a computational task is the maximum length of time the task could take to execute on a specific hardware platform. Knowing worst-case execution times is of prime importance for the calculability analysis of hard real-time systems." [1]

All the runs were performed on a machine with processor capabilities listed in 7.1. As it is shown, the used processor is of type `Intel(R) Core(TM)2 Duo CPU T7800`, which runs at `2.60GHz`. The installed operating system is an Ubuntu 8.04 - Hardy Heron distribution, released in April 2008. On this distribution, a vanilla kernel version 2.6.24 was patched with RTAI patch version 3.6-cv, and then compiled and installed. Additional parameters of the RTAI runtime are presented in the properties excerpt in 7.2. It can be observed the RTAI version is the one afore mentioned, namely `<3.6-cv>`. The version of the `gcc` compiler for compiling the RTAI sources and the additionally implemented kernel modules is `4.1.0`. The global heap size is of `2097152` bytes.

[1]http://en.wikipedia.org/wiki/Worst-case_execution_time

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 Duo CPU     T7800  @ 2.60GHz
stepping        : 11
cpu MHz         : 2593.618
cache size      : 4096 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc
arch_perfmon pebs bts pni monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr lahf_lm ida
bogomips        : 5190.66
clflush size    : 64

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 Duo CPU     T7800  @ 2.60GHz
stepping        : 11
cpu MHz         : 2593.618
cache size      : 4096 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc
arch_perfmon pebs bts pni monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr lahf_lm ida
bogomips        : 5187.37
clflush size    : 64
```

Figure 7.1: Info CPU

```
[  327.918739] I-pipe: Domain RTAI registered.
[  327.918746] RTAI[hal]: <3.6-cv> mounted over IPIPE-NOTHREADS 2.0-07.
[  327.918748] RTAI[hal]: compiled with gcc version 4.1.0.
[  327.918778] RTAI[hal]: mounted (IPIPE-NOTHREADS, IMMEDIATE (INTERNAL IRQs
VECTORED), ISOL_CPUS_MASK: 0).
[  327.918779] PIPELINE layers:
[  327.918781] f8e53a80 9ac15d93 RTAI 200
[  327.918782] c0696680 0 Linux 100
[  327.947142] RTAI[malloc]: global heap size = 2097152 bytes, <BSD>.
[  327.947526] RTAI[sched]: IMMEDIATE, MP, USER/KERNEL SPACE: <with RTAI OWN
KTASKs>, kstacks pool size = 524288 bytes.
[  327.947657] RTAI[sched]: hard timer type/freq = APIC/12411750(Hz); default
timing: oneshot; linear timed lists.
[  327.947774] RTAI[sched]: Linux timer freq = 250 (Hz), CPU freq = 2593618000
hz.
[  327.947857] RTAI[sched]: timer setup = 999 ns, resched latency = 2944 ns.
```

FIGURE 7.2: RTAI Properties And Features

The real-time scheduler is of type MultiProcessor (MP). RTAI has a UniProcessor (UP) specific scheduler, and two for MultiProcessors (MP). In the latter case you can chose between a SymmetricMultiProcessor (SMP) and a MultiUniProcessor (MUP) scheduler. In our case, the SMP processor was used. The SMP scheduler can be timed either by the 8254 or by a local APIC timer. In SMP/8254 tasks are defaulted to work on any CPU but you can assign them to any subset, or to a single CPU. For the SMP/APIC based scheduler, one can statically optimize the load distribution by binding tasks to specific CPUs. In fact, whichever local APIC is shot, will depend on the task scheduling, as that will determine the next shooting.

From 7.2 we can see that the hard timer type/freq for APIC is `12411750(Hz)` and that the rescheduling latency with RTAI is estimated to be `2944 ns`.

The strategy adopted here is to isolate each implementation of a real-time TTCN-3 specific feature and to measure its WCET latencies in benchmark scenarios. This approach has been inspired by the benchmarks developed to evaluate the timeliness of real-time features of Ada, described in [9], [8].

7.1.1 Special Operation Wait

In order to evaluate the time precision of the `wait` operation, the *TS* presented in Figure 7.3 is considered. This *TS* contains one test component whose behavior includes a part where the `wait` operation is executed in a loop. Time stamps are taken before and after the `wait` operation. The latencies are calculated comparing the time interval between those time stamps with the value of Δt, the parameter of the `wait` operation. The formula used for calculating the latency is:$\Delta\epsilon = |t_2 - t_1 - \Delta t|$, where $\Delta t \in \{100ms, 1ms, 100\mu s, 10\mu s\}$, and t_1, t_2 represents the time stamps taken instantaneously and immediately after the `wait` operation in one single iteration. There are 100 iterations for one value of the Δt parameter, meaning 100 latency samples for each.

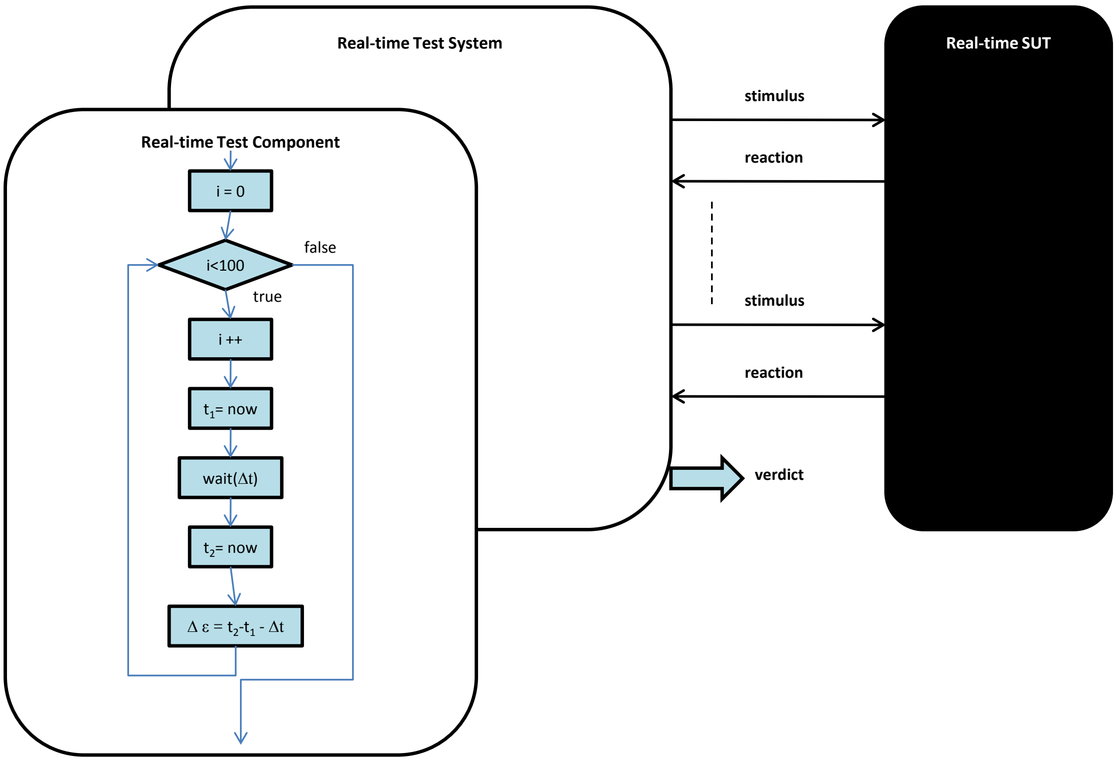


FIGURE 7.3: Test System Design For Evaluating A `wait` Operation

The Chart 7.4 presents the series with the latency variations for each value of the Δt parameter of the `wait` operation, $\Delta t \in \{100ms, 1ms, 100\mu s, 10\mu s\}$. The latencies, $\Delta\epsilon$ are measured in μs. The latency domain is represented on the vertical axis.

The horizontal axis represents the domain which indicates the number of the iteration at which a certain latency value is obtained. The latency variations represented in Chart 7.4 are small and do not overpass the limit of $3\mu s$. The conclusion that can be drawn from analyzing the chart is that we can easily state that the safe margin for the worst case execution time for the `wait` operation is of $3\mu s$. $WCET(wait100ms) = WCET(wait10\mu s) = 3\mu s$, and $WCET(wait1ms) = WCET(wait100\mu s) = 2\mu s$. These are very good latency levels for most real-time applications [7]. The experiments have been conducted for a *TS* where the test component containing the indicated behavior is the only running task in the system, or the most prioritized one.

7.1.2 Starting And Stopping Of Test Components At Precise Times

This section presents and analyzes the variation of latencies for the lifespan of a test component with preestablished start and stop execution times. The behavior of the utilized *TS* is sketched in Figure 7.5. The behavior of one test case comprised by the *TS* contains a segment which loops for an indicated number of iteration, 100 in this case, and in that loop, a test component is being started and then stopped at relative

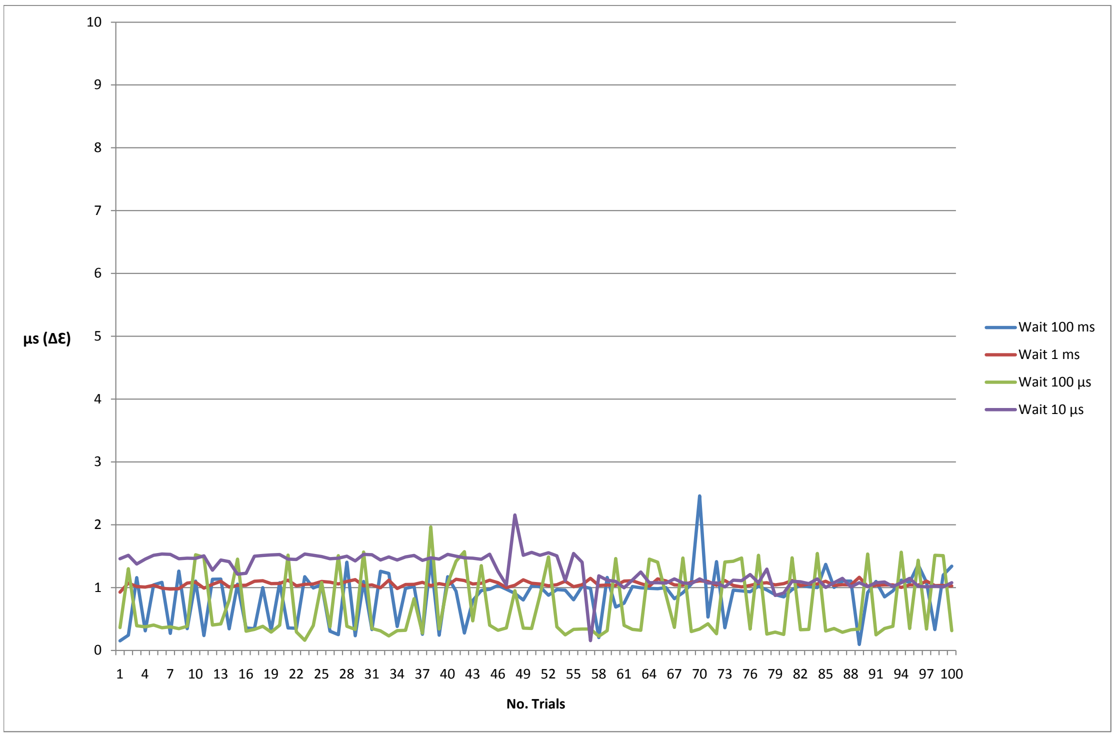


FIGURE 7.4: Latency Margins For The `wait` Operation

times, indicated by t_{i1} and t_{i2}. Those are being given as parameters to the `start at`, respectively `stop at` instructions.

On the other hand, Δt is calculated based on the time stamps taken when the component starts executing and immediately after the test component has been killed. The latency is then calculated using the formula: $\Delta\epsilon = |\Delta t - t_{i2} + t_{i1}|$. It is preestablished that the value of $t_{i2} = t_{i1} + period$, where $period \in \{100ms, 1ms, 100\mu s, 10\mu s, 1\mu s\}$ as indicated in Chart 7.6, or $period \in \{100ms, 1ms, 100\mu s\}$ as indicated in Chart 7.7. In each of the aforementioned charts, each value assigned to the *period* variable is distinguishingly defining one of the series contained in that chart.

In the previous Chapter, Section 6.1, we presented two ways for implementing the `start at`, `stop at` operations, respectively. One implementation was based on concurrent real-time tasks and the other implementation used timers. We present here two charts with latency variations, one for each choice of implementation. This way, both methods can be evaluated individually, but also through comparison to each other.

In both Charts 7.6 and 7.7 the vertical axis represents the latency scale and it is measured in μs, while the horizontal axis represents the indexes of the iterations to which the latency values are associated with. From both charts it is observable that latencies are very small, and the majority vary between 0 and 2 μs. There are only a few spikes that overpass the limit of $2\mu s$ in both charts. In the chart associated with the implementation

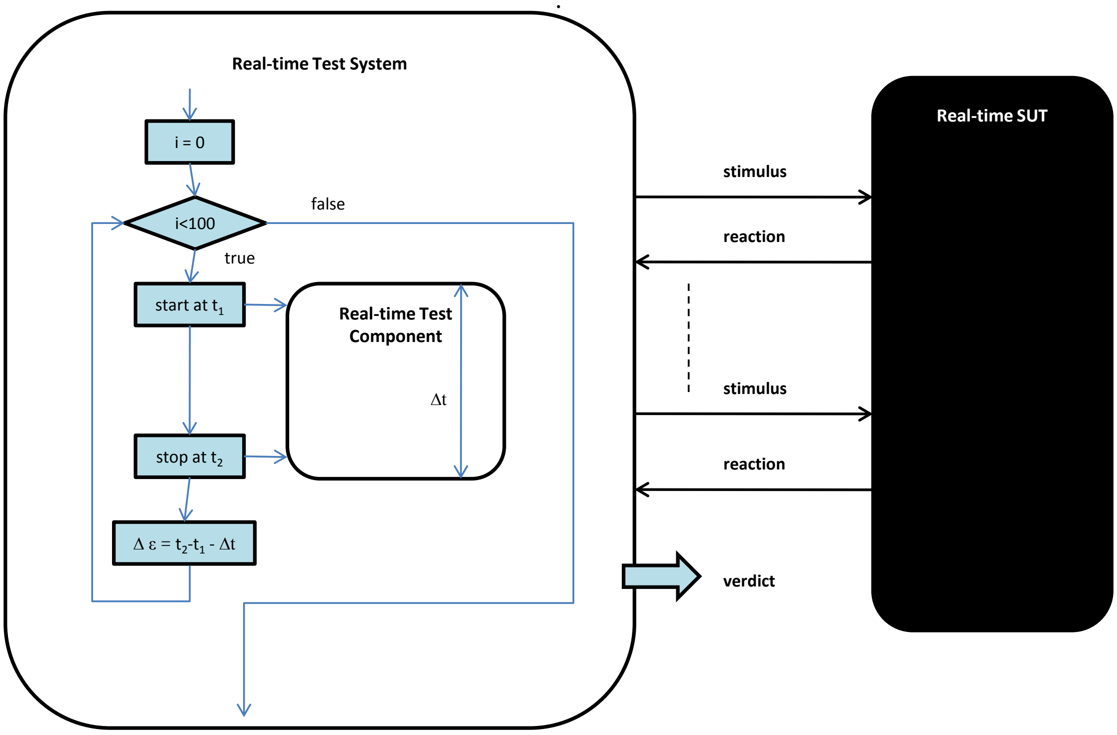


FIGURE 7.5: Test System Design For Evaluating The Precision `start at` and `stop at` Operations

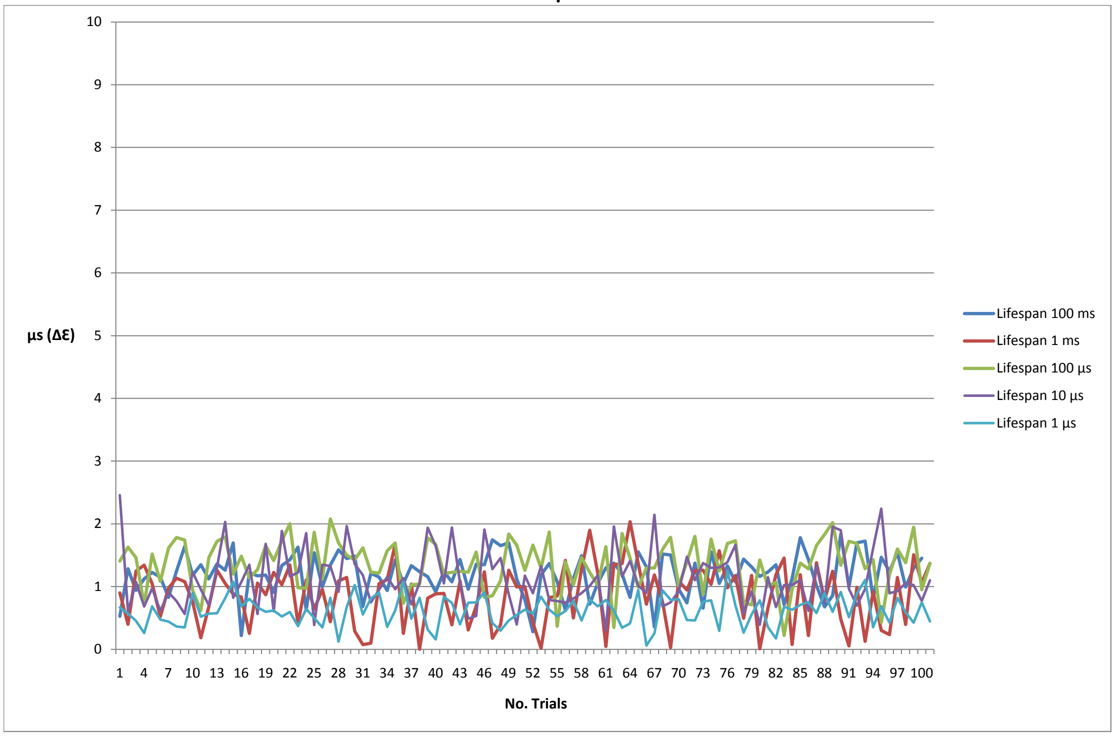


FIGURE 7.6: Latency Margins For The Lifespan Of A Test Component

involving timers, the performances are slightly better, with the majority of latencies comprised between 0 and 1 μs. The problem with the implementation using timers, was that trying to impose a test component life span of $10\mu s$ or $1\mu s$ leads to a freezing of the system. This might be due to hardware insufficiencies. The implementation with another real-time task for stopping the component also works adequately for such small life spans.

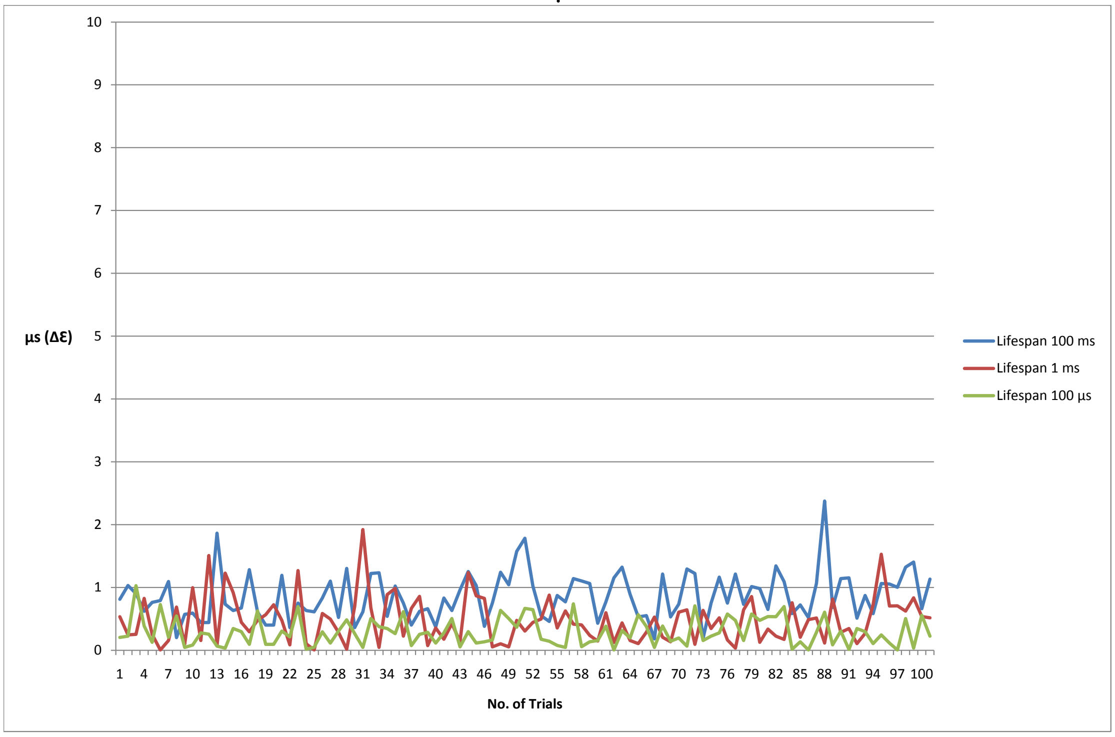


FIGURE 7.7: Latency Margins For The Lifespan Of A Test Component For The Implementation Using Timers

The results obtained in both Chart 7.6 and 7.7 were obtained from executions in which there was only one test component currently running; the same one whose lifespan is being analyzed. In this case, the $WCET(lifespan, 100ms) = WCET(lifespan, 1ms) = WCET(lifespan, 100\mu s) = WCET(lifespan, 10\mu s) = WCET(lifespan, 1\mu s) = 3\mu s$.

Chart 7.8 presents the situation when the number of test components that are required to start and stop at precise time points are gradually being increased. There were runs performed for a TS with $2, 10, 20, 30, 40, 50$ and 100 test components. For each set the latency value was mediated and the average value was put into the chart. The standard lifespan for one test component is set to $1s$. The four distinct series indicate four situations, when the components are required to start and stop at the same time points, or at time points that are distanced between one another using $10\mu s$, $100\mu s$ and $1ms$ intervals, in the following manner: $t_{start_i} = t_{start_{i-1}} + period$, where $i = 1..CompNo$, $CompNo = 1..100$ and $period \in \{0, 10\mu s, 100\mu s, 1ms\}$.

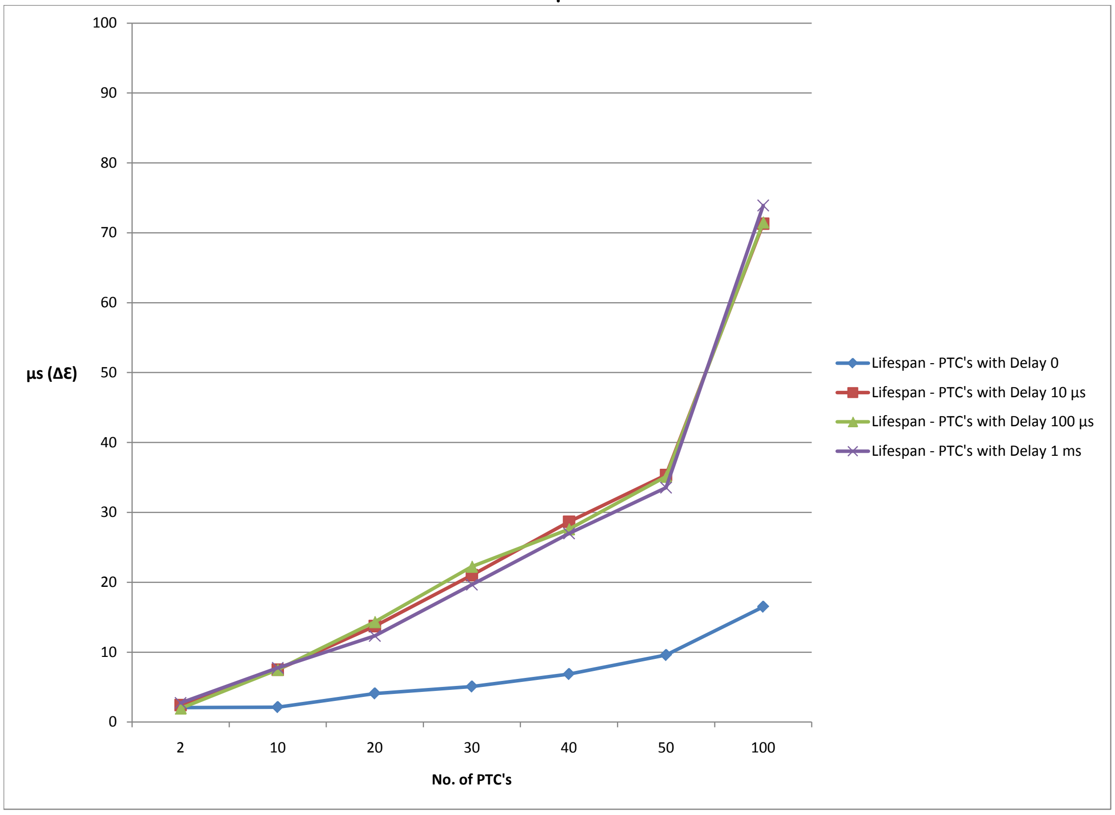


FIGURE 7.8: Latency Margins For The Lifespan Of PTC's, With An Increased Numbers Of PTC's And Variable Delays

The average values for each number of test components is calculated in each of the series, according with the formula:

$$\Delta\epsilon_{CompNo,period} = \frac{\sum_{0\le i\le CompNo} |period - t_{stop_i} + t_{start_i}|}{CompNo} \tag{7.1}$$

From Chart 7.8, it is noteworthy that it is an increasing trend of the series, that seems to be a linear function with an increase that is direct proportional with the number of test components. Therefore, we can define a $WCET$ formula, as: $WCET(CompNo) = M * CompNo$, where $M > 1$ is a constant, and $CompNo = 1...100$. We did not perform tests with a number of components greater than 100, and we can not guarantee for the behavior of the TS for these values. It is assumed nevertheless, that tests performed for testing embedded systems should be kept simple and highly specialized on some specific parts of behavior. Therefore, a high number of test component running concurrently would not be required.

7.1.3 Sending Messages At Precise Times

In this section, the latencies for the `send at` operation are going to be analyzed and discussed. The test setup is the one presented in Figure 7.9. The behavior of one test component is going to be enhanced by a loop comprising a `send` operation. The test component is implemented as a periodic real-time task, which becomes active only when the time comes for the `send` operation to be executed. After the `send` is performed, the task yields processor, until the next activation time. The values of the time points when the `send` operation is to be performed are respecting the succession: $t_i = t_{i-1} + \Delta t$. If we record the time stamps when the `send` operation is actually being performed, the calculation of the latency is done according to the formula: $\Delta\epsilon_i = |t_{timestamp_i} - t_{timestamp_{i-1}} - \Delta t|$.

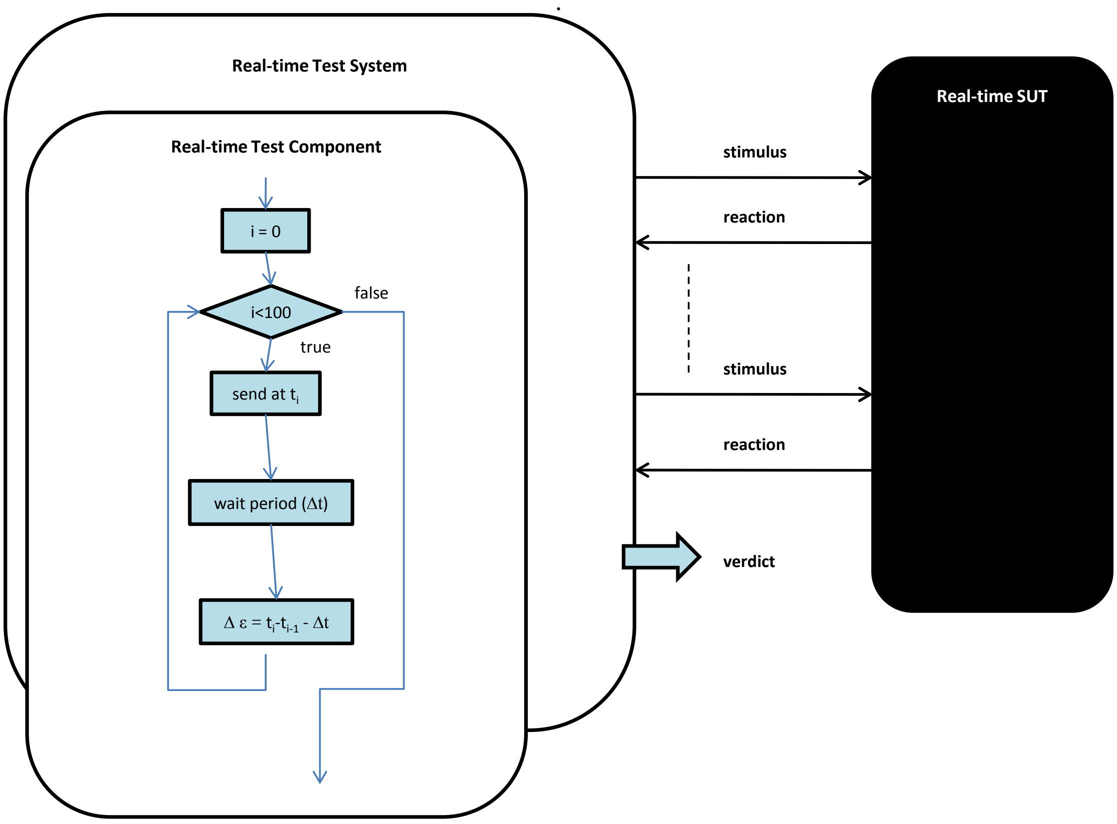


FIGURE 7.9: Test System For Evaluating The `send` Operations With Strict Timings

As discussed in Section 6.1 of the previous chapter, there are two proposed implementations for a `send at` operation; one realized only with concurrent real-time tasks, and the other one using real-time timers. Charts 7.10 and 7.11 show the experimental results for both tasks and timer implementations respectively. In both charts, the send operation was performed through 100 iterations. For each chart we have five distinct series. Each series represents the variations of `send at` latencies for a predefined delay period, Δt, of $\{100ms, 1ms, 100\mu s, 10\mu s, 1\mu s\}$.

It can be noted that by comparing the two charts, the performances for the implementation with timers are slightly better than the results for the implementation using only a real-time task. This can be explained due to the fact that the timers are implemented as tasklets, so the changing of context is performed more quickly. Nevertheless, for very

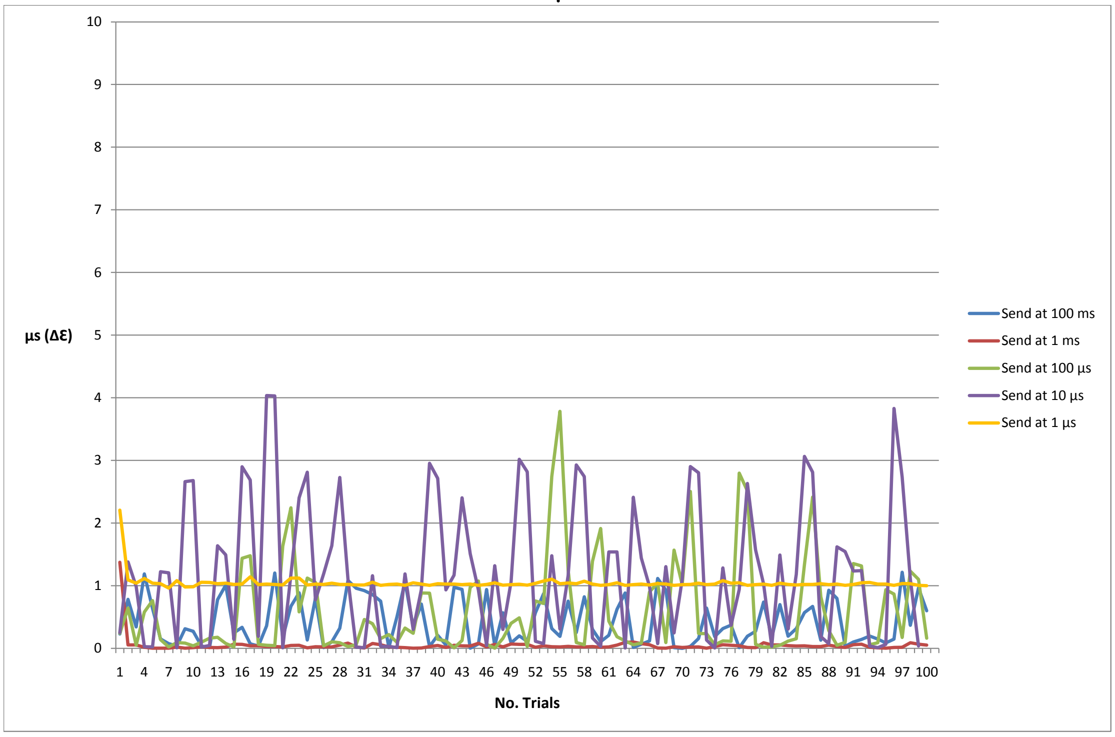


FIGURE 7.10: Latency Margins For The `send` Operation With Strict Timing

small periods, as for example $1\mu s$, the implementation with timers starts to perform worse than the implementation with tasks.

From the two charts we can draw the conclusion that the $WCET_{tasks}(sendat100ms)$, $WCET_{tasks}(sendat1ms)$, $WCET_{tasks}(sendat100\mu s)$, $WCET_{tasks}(sendat100\mu s) \le 4\mu s$ and $WCET_{timers}(sendat100ms)$, $WCET_{timers}(sendat1ms)$, $WCET_{timers}(sendat100\mu s)$, $WCET_{timers}(sendat100\mu s) \le 3\mu s$.

In the following Charts, 7.12 and 7.13, there is an analysis of the situations when the periodic component holding the `send at` operation is not the only one available to run in the *TS*, but an increased load of other components is added as well. In Chart 7.12 the added test components are translated to real-time tasks having the same priority with the task holding the `send at` operation. In Chart 7.13 the added test components are translated to real-time tasks having a lower priority than the task holding the `send at` operation.

In the case that all the running components have been set with equal priority, an ascendent trend can be noted. The latencies become increasing linear with the number of test components being added. The latency values on the Charts 7.12 and 7.13 represents average values that are calculated based on individual latencies for 100 `send at` operations executed in different load conditions - $2, 10, 20, ..100$ added test components - and different predefined periods between them - $\Delta t \in \{100ms, 1ms, 100\mu s, 10\mu s\}$. The

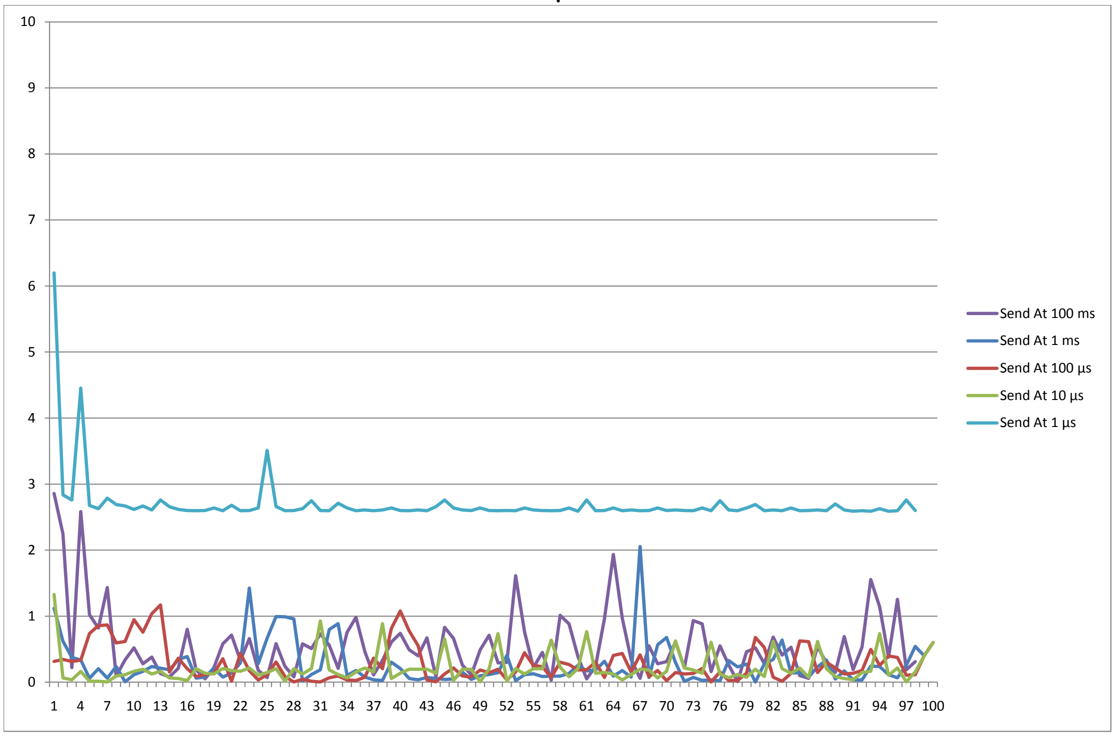


FIGURE 7.11: Latency Margins For The `send` Operation For The Implementation Using Timers

formula used for calculating the average value is:

$$\Delta\epsilon_{CompNo,period} = \frac{\sum_{0\leq i<100}\Delta\epsilon_i}{100} \tag{7.2}$$

It can be noticed that in the situation when the priorities for the additional test components are less than the priority of the component holding the `send at` operation, the ascendent trend disappears, and the latency variations dwell between $4\mu s$ and $5\mu s$. This means that the latency variations for this case are hardly influenced by the load, and the $4\mu s \leq WCETs_{load}(sendat) \leq 5\mu s$.

For the situation where the priorities for the additional test components equal the priority of the component holding the `send at` operation, the trend being linear ascendent, we can approximate $WCETs_{load}(sendat, CompNo) \leq M * CompNo\mu s$.

7.1.4 Receive Operation Limited By The Break Instruction

In this section we are going to discuss the time precision of a `receive` operation which is limited by a `break at` condition. The structure is as presented in Figure 7.14. There is a test component that activates a real-time task to wait on a port for a specific message. Together with the activation of the receive task, the component is responsible for also initializing another task which simply waits for a predefined period of time, Δt. If the message is received before the Δt interval expires, then the receive task kills the break

.

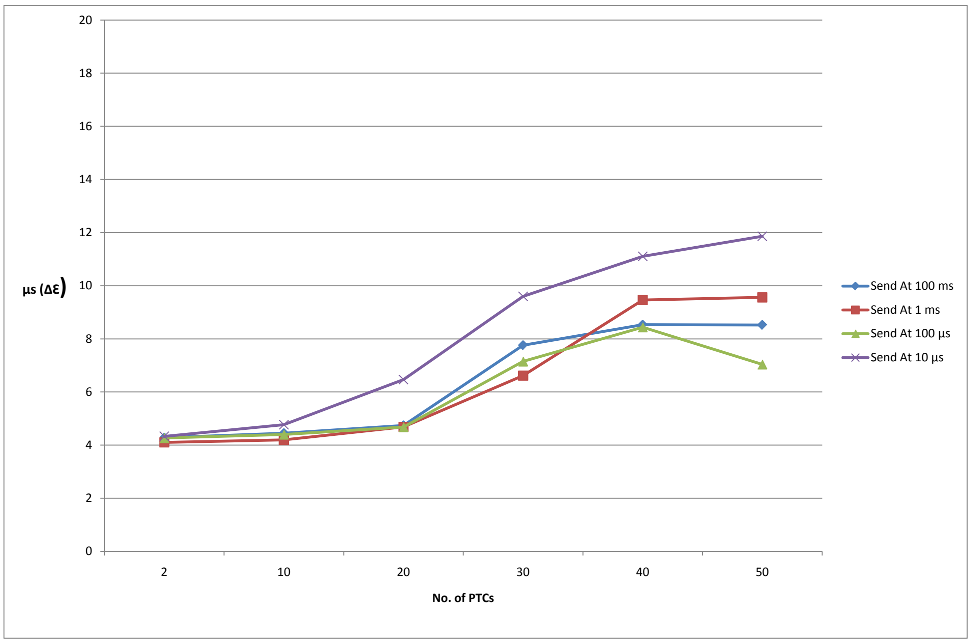


FIGURE 7.12: Latency Margins For The `send` Operation With Increasing Number Of PTCs In The Background. Same Priority.

.

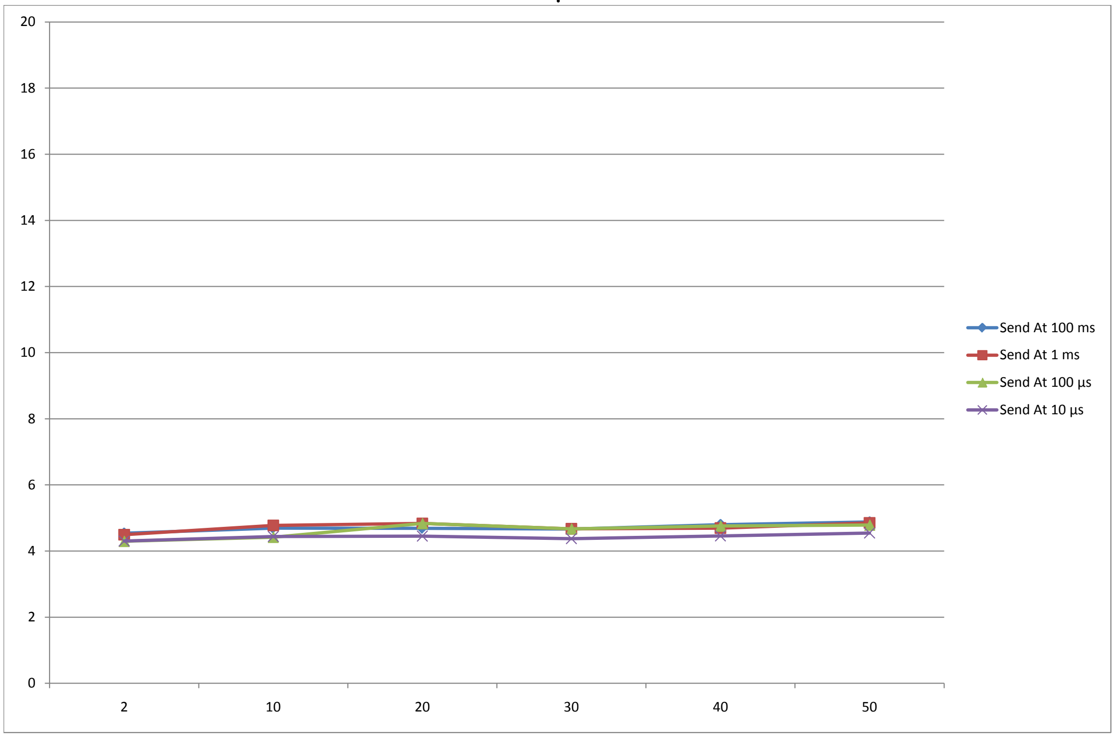


FIGURE 7.13: Latency Margins For The `send` Operation With Increasing Number Of PTCs In The Background. Greater Priority For `send` Operation.

task. Otherwise, it happens the other way round. When the Δt interval expires, the break tasks kills the receive task. In order to be sure that the break task will function properly we measure the time points immediately when receive starts, and immediately after an unsuccessful receive task is finished. Based on these recorded time stamps, the time interval is measured and compared to the predefined one, according to the formula:

$\Delta\epsilon = |t_{killed\_receive} - t_{started\_receive} - \Delta t|$.

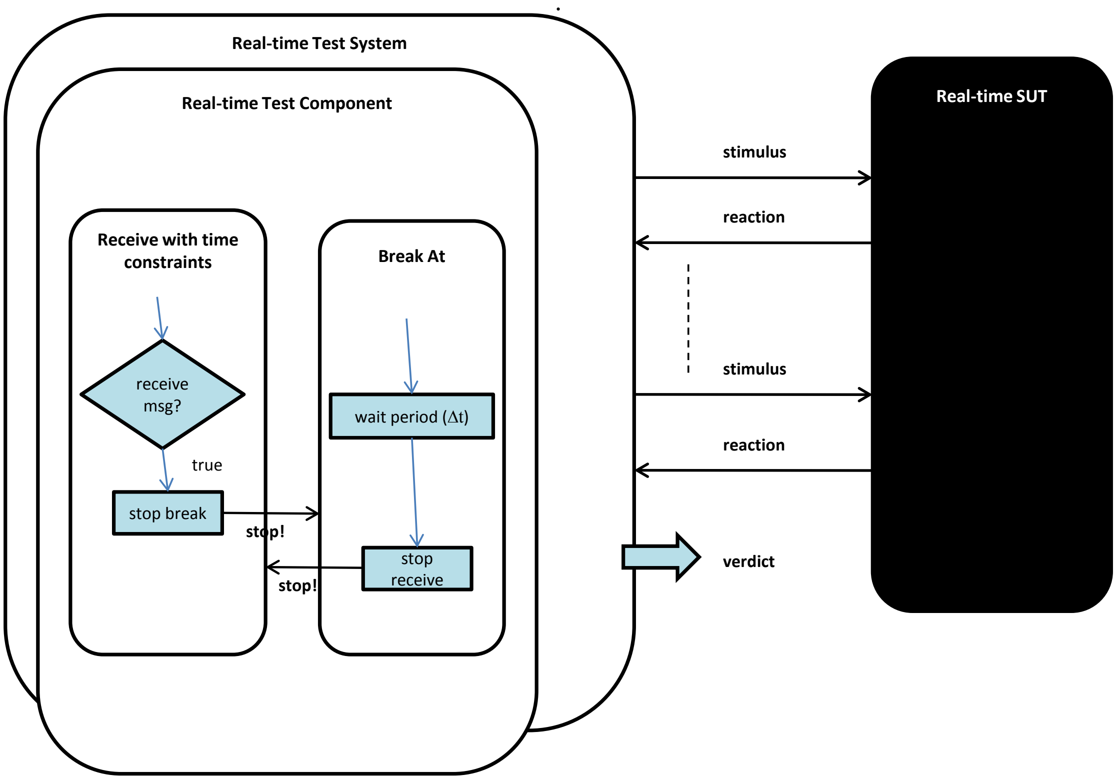


FIGURE 7.14: Test System Structure For `receive...break` Evaluation

The Charts 7.15 and 7.16 present the experimental results for the latency variations. The implementation with real-time tasks is evaluated in the case of Chart 7.15 and the implementation is based on timers evaluated in the case of Chart 7.16. The latency variations are calculated for $\Delta t \in \{1sec, 100ms, 1ms\}$ predefined `break at` periods. For each Δt period, the `receive` with `break at` operation is performed 100 times. Comparing the two charts, we can notice that the implementation with timers have better latencies that the one using tasks. This might be due to the fact that the timers are implementing with tasklets, and the changing of context for these takes place more rapidly. From the two charts we may conclude that $WCET_{tasks}(receive..breakat1sec) = WCET_{tasks}(receive..breakat100ms) = WCET_{tasks}(receive..breakat1ms) \leq 4\mu s$ and $WCET_{timers}(receive..breakat1sec) = WCET_{timers}(receive..breakat100ms) = WCET_{timers}(receive..breakat1ms) \leq 2\mu s$

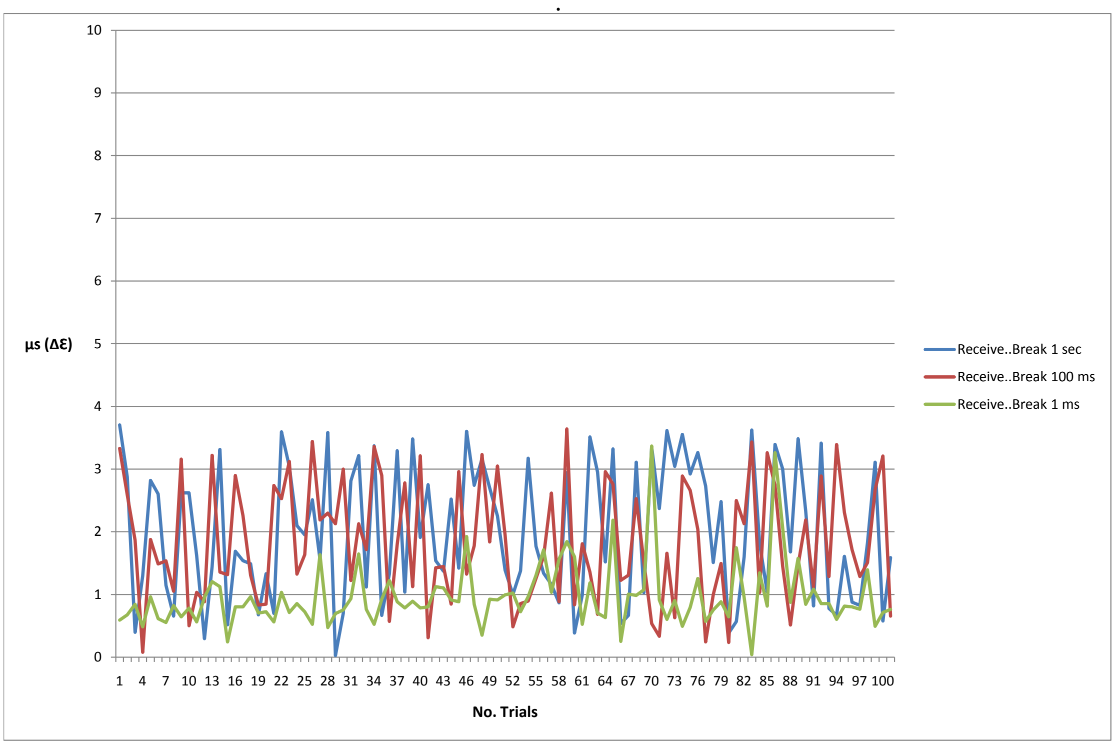


FIGURE 7.15: Latency Margins For The `break` Instruction

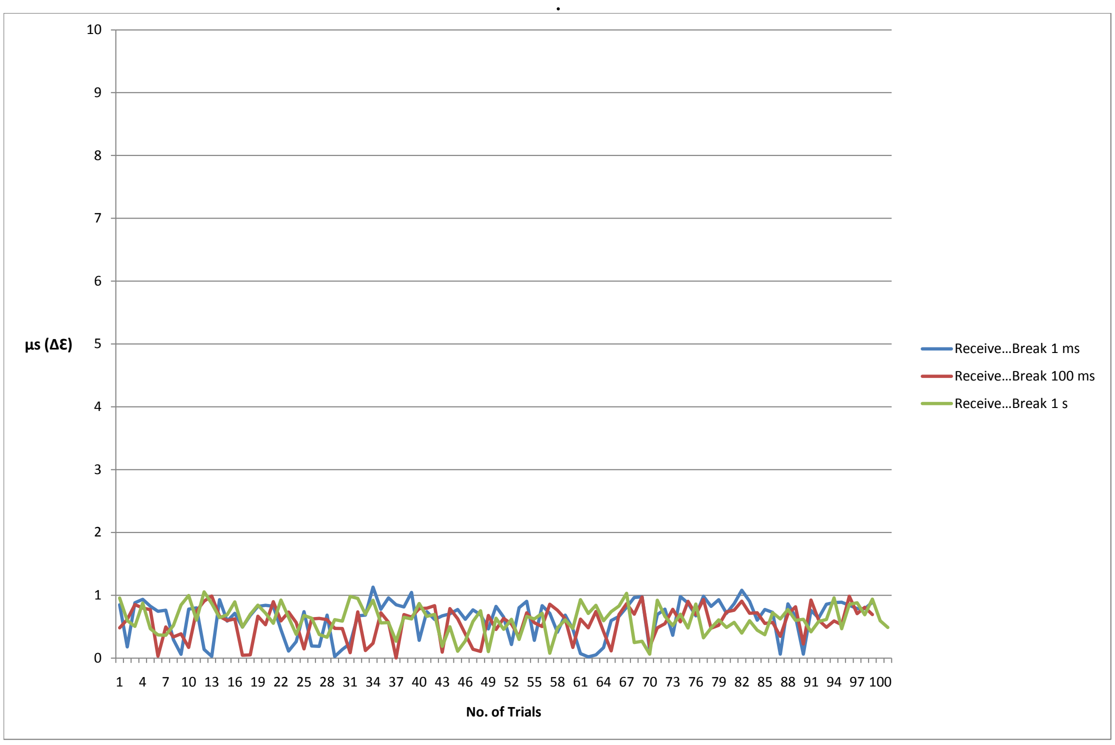


FIGURE 7.16: Latency Margins For The `break` Instruction Implemented With Timers

7.2 Results For The Auto Car Door Case Study With A Test System Implemented On FreeRTOS

This section presents the results for the test case study whose implementation, based on FreeRTOS, has been presented in Section 6.2 of Chapter 6. This example is interesting because it shows all the concepts in combination and the *RTTS* can be evaluated as a whole, in a real life scenario, against a real-time *SUT*.

Figure 7.17 illustrates how the flash signal should look like in order to respect the tested time requirement that is tested with the test specification listed in Listing 6.32 from Section 6.2. This time requirement is stated in the following:

> "The flash light remains "ON" for no more than 500 clock ticks; or the interval between a consecutive ON and OFF response is no longer than 500 clock ticks."

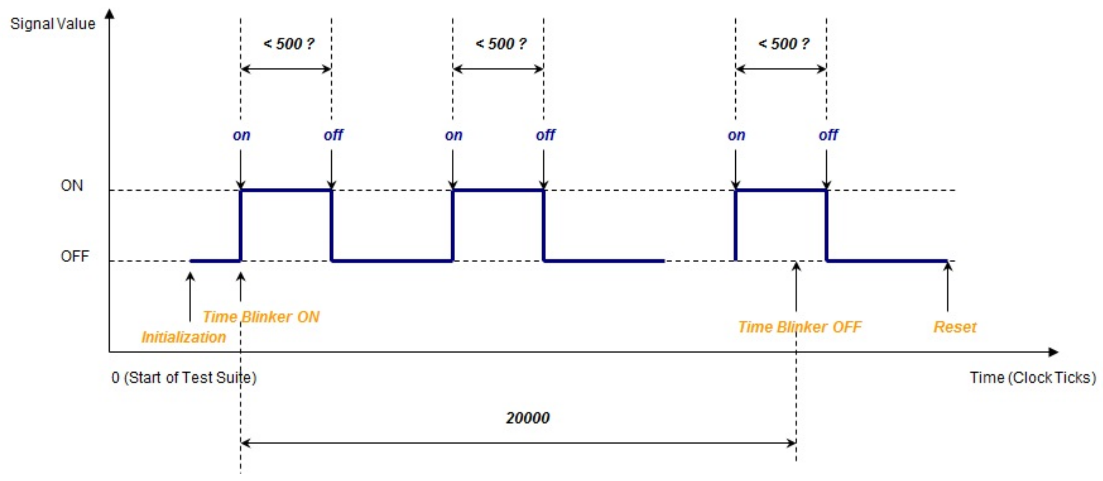
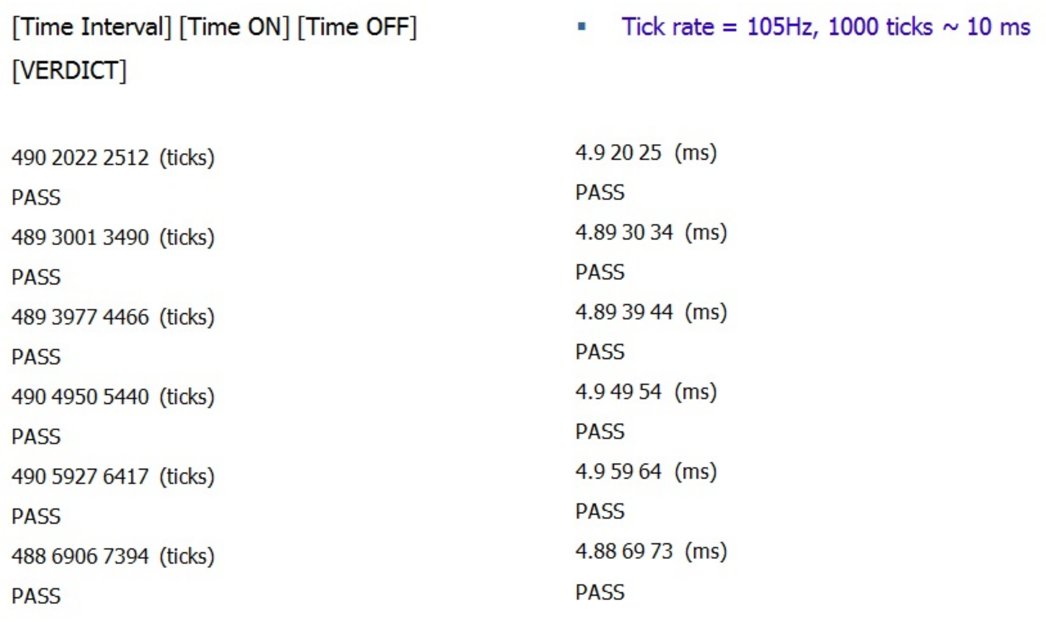


FIGURE 7.17: One Possible Flash Light Signal Flow

Considering that the tick rate is of 10^5 Hz then 1000×ticks would be the equivalent of 10×ms. Figure 7.18 presents an excerpt from a list of results, indicating time intervals between consecutive "on-off" messages received from the *SUT* between one "Time_Blinker_ON'" command and one "Time_Blinker_OFF" command from the *TS*.

If the time requirement is changed to a time frame of 200 clock ticks, in order to reflect the increased frequency between "on"-"off" messages when a "Panic_Blinker_ON" command has been issued, the signal will look as illustrated in Figure 7.19. In order to see if the *TS* assigns correct verdicts, the behavior of the *SUT* will be influenced after a while, to stop the panic blinking, through the issue of a "Panic_Blinker_OFF" command, and will start a slower paced blinking in between "Time_Blinker_ON" and "Time_Blinker_OFF" commands. The requirement of 200 clock ticks, equivalent to



```
[Time Interval] [Time ON] [Time OFF]          ▪ Tick rate = 105Hz, 1000 ticks ~ 10 ms
[VERDICT]

490 2022 2512  (ticks)                        4.9 20 25  (ms)
PASS                                          PASS
489 3001 3490  (ticks)                        4.89 30 34  (ms)
PASS                                          PASS
489 3977 4466  (ticks)                        4.89 39 44  (ms)
PASS                                          PASS
490 4950 5440  (ticks)                        4.9 49 54  (ms)
PASS                                          PASS
490 5927 6417  (ticks)                        4.9 59 64  (ms)
PASS                                          PASS
488 6906 7394  (ticks)                        4.88 69 73  (ms)
PASS                                          PASS
```

FIGURE 7.18: A Sample Of Case Study Results

2×milliseconds, will not be changed and therefore, the new "on"-"off" time distance measurements will be interpreted as failures by the *TS*.

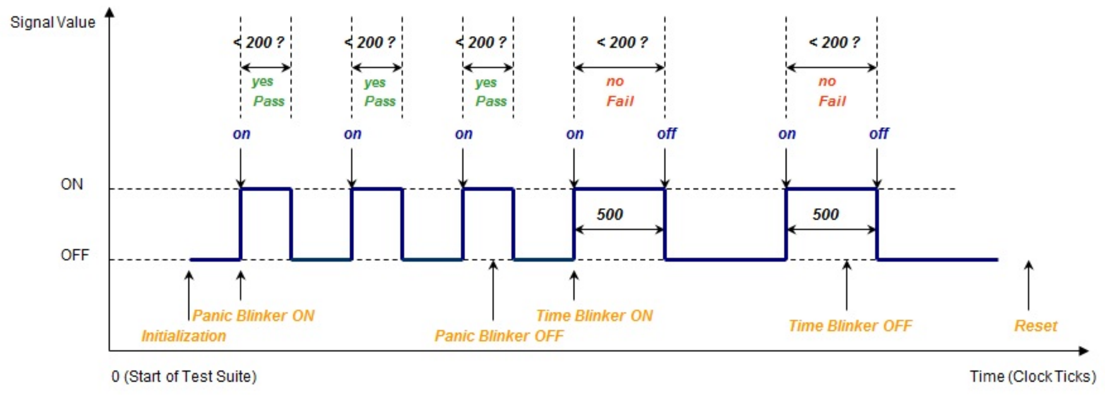


FIGURE 7.19: Another Possible Flash Light Signal Flow

By combining different time restrictions with different sets of commands sent to the controller, different timed behaviors of the *SUT* were analyzed. Based on the results we could estimate the precision of the *SUT* down to the range of a hundreds of a microsecond. At finer granularity, delays were found, caused by buffered reading on local ports of the controller. By concatenating and sending long combined control messages, we received failure, in addition to some of the commands being lost before being read, as was the case if they exceeded the capacity of the port buffer.

The time between sending a command and receiving the response message from the *SUT* was also measured. The total time included, beside the reaction time of the *SUT*, also the time that both the command-message and the response-message had spend traveling

TABLE 7.1: The Results For The Presented Example

Sent_at (ticks)	**Received_at (ticks)**	**Interval (ticks)**	**Constraint (ticks)**	**Verdict**
2000	2002	2	3000	**pass**
3000	3001	1	3000	**pass**
4000	4001	1	3000	**pass**
5000	5118	118	3000	**pass**
6000	6002	2	3000	**pass**
7000	7001	1	3000	**pass**
8000	8031	31	3000	**pass**
9000	9001	1	3000	**pass**
10000	10001	1	3000	**pass**
11000	11541	541	3000	**pass**

on the medium. In this case, the medium is the serial cable. For a tick rate of 10^5 Hz, we obtained the numbers listed in Table 7.1. All the tests were passed. The table contains the relative times, measured in the number of clock ticks for sending and for receiving the message. The real-time constraint was 3000 ticks which means 30×milliseconds.

7.3 Summary

The benchmark of the implementations based on Linux with RTAI showed that their delays and latencies are situated in bounded intervals in the range of microseconds. These are very good latency levels for most real-time applications [7]. In practice, FreeRTOS also proved to work very well as a small and purpose-oriented real-time operating system for testing simple real-time applications. The FreeRTOS implementation was precise and good at discovering faults of the *SUT*. The latencies and delays were upper bounded in the range of microseconds, which was good, but not surprising for a small real-time operating system. This proves that the realization of concepts, using both FreeRTOS and Linux with RTAI, serves the goal of this work. The realization of frameworks for real-time testing that are reliable on a microsecond level has been achived.

Chapter 8

Conclusion

"Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience."
Roger Bacon

As this is the final chapter of this thesis, a short summary of the entire work will be made and the new perspectives, opened through this research, will be emphasized.

8.1 Summary Of The Thesis

The solution provided in this thesis was concerned with the design and implementation of a testing framework for real-time embedded applications. The aim is to provide a standard-based test technology that could be successfully used for automating the test procedures, especially with regard to the real-time aspects, in domains with rapid development process and high quality demands like those of the automotive industry.

As stated already in Chapter 1 of this thesis, the applicability and usage of embedded systems in industry has encountered a considerable growth during the past years. This growth was recorded in many areas such as: automotive, avionics, power plant control, robotics, etc., where functionality based on embedded controllers represents a high percentage of the overall functionality. For all these domains, the functionality of real-time and embedded applications is usually coupled with time requirements that must be strictly respected.

The boost in application diversity, together with the increased complexity of requirements, has triggered a need for a reliable testing procedure, with a new focus: instead of testing only the functional aspects, the need for testing time-related features emerged.

Different proprietary testing technologies emerged as a response to that need. Distinct individual solutions, with their main focus on the time and performance aspects, were created in order to test real-time applications. Some of these solutions where completely hardware-based, some were only partial automatized, some were targeting only particular types of systems. Their evolution was heterogenous and the demand for a common and standardized approach has emerged.

The real-time testing methodology and framework presented here are based on a standardized test language that was proven to be popular and successful in the industry, in areas such as mobile and broadband telecommunications, medical system, IT systems and, more recently, in the automotive industry. The referred language is Testing and Test Control Notation version 3 (TTCN-3), developed and maintained by European Telecommunications Standards Institute (ETSI). Having the advantage of being a well modularized, test-oriented, user friendly and popular, TTCN-3 has also the downside of not being developed with real-time focus in mind. Thus, it lacks a particular mechanism for dealing with real-time specific test situation. The insufficiencies of TTCN-3 language towards real-time were discussed in detail in Section 4.2.

After investigating the requirements specific to real-time applications and the problems that arose due to the particularities of those requirements, a short introspection into the world of real-time programming languages was made. The features that are introduced by real-time programming languages to produce real-time behaviors were analyzed in Section 2.1.2. Also, several past attempts of improving TTCN-3 language with new semantics were referenced and analyzed in Section 3.3. Based on this knowledge, new concepts for real-time were introduced in TTCN-3 in Chapter 4. These concepts represent the fundament of the present solution. Some of them were developed in collaboration with the TEMEA [79], a project on the basis of which, the additional standardized extension for TTCN-3, regarding real-time and performance, was published [80].

The semantics of the real-time test system realized on the basis of enhanced TTCN-3 was defined in Chapter 4, Section 4.5.1, by means of timed automata. For each TTCN-3 instruction that relies on a real-time concept, the associated semantic is represented as TA. This approach is new and different from the way semantics of TTCN-3 was previously defined into the standard. The motivation for choosing TA is that they are mathematical instruments specialized in modeling timed behavior in a formal way. A short introduction of TA is provided in Chapter 2, Section 2.1.4. This approach also opens new and interesting possibilities, as, for example, semantical validation of the timed TTCN-3 code, based on model-checking methods developed for timed automata. A more detailed discussion about model-checking with TA is provided in more detail in Section 2.1.4. In Appendix A the newly proposed concepts are integrated into the syntactical structure of the TTCN-3 language, by means of clear syntactic rules, based on extended Backus–Naur Form (ESBNF) notation.

After the syntactical and semantical definition, the algorithmic aspects involved for implementing the real-time test platform based on these concepts were presented in Chapter 5. The design of a real-time architecture has also been provided in this chapter. The purpose of this architecture is to logically draw a connection between abstract test concepts and real-time operating system mechanisms, in a generic way. This means, by having this perspective, theoretically any real-time operating system presenting the required mechanisms could realize the behavior implied by a certain real-time test concept. For realizing a concept proof of this approach, a number of the most representative real-time operating systems were investigated in Section 5.2. As a result of the selection process, two of those operating systems were chosen for the empirical implementation of the concepts. Those two operating systems have been presented in Sections 5.3 and 5.4.

In Chapter 6, the mappings between the real-time concepts and actual implementations on the two chosen platform have been provided. The implementation of the concepts is structured in such a manner that it can be easily integrated, by means of utility functions, into the TCI [63] and TRI [62] of TTCN-3.

The mapping implementations of concepts on the two chosen platforms were evaluated in Chapter 7. Worst case execution times (WCETs) were computed for the introduced concepts. The concepts were tested in different contexts of usage. The time determinism of the *TS* was measured by pinning time evaluation points at the beginning and ending of instructions. The obtained results also reflect the time characteristics of the chosen real-time operating systems such as the delays and the latencies. The results also proved that the approach works and the implementation of various real-time tests can be accomplished with a guarantee of them respecting deadlines in the order of microseconds. This guarantee is provided for a *TS* that is having a behavior composed by a set of tasks that respects the schedulability test for the used scheduling algorithm.

8.2 Main Contributions

The main contribution of this thesis is that it has provided a complete kit for building a standardized and automatic testing framework for real-time systems. The kit, or solution, has been revealed as the chapters of this thesis have progressed, from the introduction of abstract real-time concepts that are integrated into a chosen standardized testing language, to specific implementations of these concepts on concrete real-time platforms.

A standardized testing language was needed to provide the possibility of describing tests in an easy way, a way that can be used and understood without difficulty among different stakeholder in the industry. We chose TTCN-3 as a well-designed testing language with a high degree of popularity and usage in the real world. One challenge of this work was to identify the aspects regarding the real-time testing where TTCN-3 was not expressive enough, and to then cover these situations by introducing a minimal set of concepts that will not burden, but rather enhance the language. The set of concepts presented here originates from the set of extensions developed in the context of TEMEA project, in which the author of this thesis was involved [79]. Part of these concepts were incorporated into the new standard extension [80].

Another highlight of the thesis was that each new TTCN-3 extension has been endowed with a clear semantic, defined by a well-constructed mathematical formalism. Timed automata were used to model the introduced concepts and the test system itself. The semantic developed here established mappings between TTCN-3 abstract test specifications and timed automata, the conversion being made possible in both directions.

Based on the description of behavior in the previously defined formalism, mappings between real-time TTCN-3 concepts, and two real-time operating system were realized. Based on the experience of these two implementations, design patterns were derived.

Delays and latencies were measured for each implemented extended TTCN-3 language feature in benchmark scenarios. Evaluation of the results led to the conclusion that the goal was reached. We managed to develop a test framework for real-time, based on TTCN-3, that has WCETs bounded in the range of microseconds for basic testing behavior, comprising of a real-time schedule-able set of tasks. These results are very good, as most of real-time applications have timed requirements in the range of tens of microseconds. The maximum latencies measured through benckmarking, show that the *RTTS* is satisfying the general required timeliness and is appropriate for testing a wide range of real-time applications.

A case study was conducted as well, to show the applicability of the approach in a small real-world example. In this case study, the *SUT* consisted of an ECU controlling the functionality of a car door. Differently triggered sequences of functionality were required to be performed with respect to strict time constraints. The real-time test framework implemented on the basis of a FreeRTOS operating system was proved to be efficient in asserting conformance to both functional and timing requirements, with a precision, for the timing requirements, in the range of microseconds. Some failures in the behavior of the *SUT* were also discovered for specific sets of inputs. Thus, a proof of concept was successfully accomplished.

8.3 Future Work

Building a testing framework for real-time systems is, nevertheless, a very complex topic to cover completely at one time. The solution is meant to represent a fundament on which further research should be made. Some ideas for further directions of research will be presented in the following section.

The mappings from concepts to code, together with the associated syntax and semantics of these concepts, could be used as a fundament for implementing a compiler, able to generate platform-specific code from the real-time abstract test specification. In addition to this, within the scope of this thesis, only two real-time operating systems were chosen for the proof of concept. There are a wide range of real-time operating systems out there for selection. Each RTOS provides targeted features for specific classes of real-time applications. One compiler that could also generate some platform-independent code, and the possibility of using the generated code with specific libraries that are implementing functionalities specific to individual platforms would provide a solution to homogenize the diversity. This could be of great use to a community of real-time developers interested in extending the solution.

The step of implementing such a compiler was considered outside the scope of the thesis. The purpose of this thesis was to provide the set of concepts for real-time testing, together with a clear semantics and to prove that these concepts can be implemented on concrete real-time platforms.

Because the semantics of the new real-time features which were added to the TTCN-3 language is expressed using TA, an interesting idea would be to implement a translator from a real-time test specification to a network of timed automata. This would open the possibility of applying model-checking verification techniques to semantically validate the real-time test specification. If an automatic translation from TTCN-3 to timed automata can be performed, the generated timed automata model can be used as an input for an already existing verification tool, such as Uppaal [134], for example, that can be used to perform this type of check.

Another future aim of this thesis would be that the set of extensions introduced here will be fully incorporated into a new standardized extension for TTCN-3.

8.4 Closing Words

Real-time testing is a hot topic now days and has a huge potential of applicability in a wide range of domains. This work brings its contribution in the field of standardized and automatized testing for real-time by defining a thorough methodology and a specialized set of instruments and examples for building a framework for this type of testing. Thus, considering that the goal presented in the starting chapter of this thesis was achieved, the greatest aim of this work is to be continued, extended and, most importantly, applied, in all types of industrial situations.

Appendix A

Syntax For The Real-time Extensions Of TTCN-3

This Appendix contains the syntax for the real-time extensions of TTCN-3 that were introduced in Chaper 4.

For defining the grammatical structure for the proposed concepts, we use the extended Backus–Naur Form (BNF) notation, the same one being used by the TTCN-3 specification [2]. The symbols of the meta-notation are summarized in Table A.1. Table A.2 lists the newly introduced concepts into four columns: the first column introduces the special operations, the second column introduces the temporal predicates, the third column introduces the predefined constants used to build `timespan` values, and the last column lists the newly introduced data types.

TABLE A.1: The Syntactic Meta-notation

`::=`	is defined to be
`abc xyz`	abc followed by xyz
`|`	alternative
`[abc]`	0 or 1 instances of abc
`{abc}`	0 or more instances of abc
`{abc}+`	1 or more instances of abc
`(...)`	textual grouping
`Abc`	the non-terminal symbol abc
`"abc"`	a terminal symbol abc

TABLE A.2: List Of RT-TTCN-3 Terminals Which Are Reserved Words

`now`	`at`	`hour`	`datetime`
`wait`	`within`	`min`	`timespan`
`break`	`before`	`sec`	`tick`
`timestamp`	`after`	`millisec`	
`testcasestart`	`not`	`microsec`	
`testcomponentstart`		`nanosec`	
`testcomponentstop`			

In the following sections, the syntactical rules used for integrating the new concepts into the grammar of TTCN-3 are presented.

A.1 Data Types Suitable For Expressing Time Values: `datetime`, `timespan`, `float`, `tick`

The syntax presented in this section comprehends the rules for integration of the proposed data types, and their values into the grammar of TTCN-3. The syntactic definitions should be regarded as a formal guidance of how the new data types and values should be incorporated into the already existing structure of the language, but they also provide a view of how they relate towards already existing data types and values.

All the formal definitions are accompanied by an example of usage for a better understanding of the theory.

PredefinedType::= BitStringKeyword | BooleanKeyword | CharStringKeyword |
IntegerKeyword | VerdictTypeKeyword | FloatKeyword |
DateTimeKeyword | TimeSpanKeyword | TickKeyword ;

DateTimeKeyword::= ***"datetime"***;
TimeSpanKeyword::= ***"timespan"***;
TickKeyword::= ***"tick"***;

PredefinedValue ::= BitStringValue | BooleanValue | CharStringValue |
IntegerValue | VerdictTypeValue | FloatValue |
DateTimeValue | TimeSpanValue | TickValue;

The format of `datetime` values is ISO 8601:2004 [112] - compliant, and the grammar rules for building the values are presented in the following section.

Nevertheless, the focus of the thesis has been on the real-time embedded systems, that don't imply any distribution and where the timed functionality for investigation evolves at a micro time scale. Therefore, the emphasis will be on the other time data types.

DateTimeValue::=[FullDate]***"T"***PartialTime;

FullDate::=(Year***"-"***Month[***"-"***Day]) | (Year[***"-"***Month***"-"***Day]);

PartialTime::=[PartialBig]***"_"***PartialSmall;

PartialBig::=([Hour***":"***]Minute***":"***Second) | ([Hour***":"***Minute***":"***]Second);

PartialSmall::=Millisecond***":"***Microsecond***":"***Nanosecond;

Day::=Number; **Month::=**Number; **Year::=**Number; **Hour::=**Number;
Minute::=Number; **Second::=**Number; **Millisecond::=**Number;
Microsecond::=Number; **Nanosecond::=**Number;
Number::=(NonZeroNum{Num}) | ***"0"***;
Num::=***"0"*** | NonZeroNum;
NonZeroNum::=***"1"*** | ***"2"*** | ***"3"*** | ***"4"*** | ***"5"*** | ***"6"*** | ***"7"*** | ***"8"*** | ***"9"***;

`timespan` data value is expressed as a composite expression in which the terms are represented by float values multiplying predefined units. The domain of `timespan` values can be considered an ordered set of durations, denoted as $\mathbb{TS}$. The durations represent intervals of time that spans between events.

The indicated `"nanosec"`, `"microsec"`, `"millisec"`, `"sec"`, `"min"`, `"hour"` are predefined time units which are going to be defined as tool's constants. We are targeting tests with the time granularity refined to the nanoseconds level, therefore we are providing means to explicitly express time intervals up to that micro level.

TimeSpanValue::=TimeSpanValue *"+"*TimeSpanTerm | TimeSpanTerm**;**

TimeSpanTerm::=FloatValue *"*"* NanosecKeyword | FloatValue *"*"* MicrosecKeyword |
FloatValue *"*"* MillisecKeyword | FloatValue *"*"* SecKeyword |
FloatValue *"*"* MinKeyword | FloatValue *"*"* HourKeyword**;**

In this context, the syntax of `FloatValue` is a simplification of the one in the standard.

FloatValue::=Number[*"."*DecimalNumber]

DecimalNumber::={Num}+

NanosecKeyword::= ***"nanosec"***; **MicrosecKeyword::=** ***"microsec"***;

MillisecKeyword::= ***"millisec"***; **SecKeyword::=** ***"sec"***;

MinKeyword::= ***"min"***; **HourKeyword::=** ***"hour"***;

TickValue::=Number;

`tick` data type represents, at the test specification level, the internal count tick number of the CPU and it is actually a positive integer initialized at the starting of the system. The internal count tick number is directly proportional to the frequency of the CPU. This data type is introduced to provide insight of the timing features of the physical machine (e.g. ticks per second). Since we are dealing with real-time constraints for the *SUT* which are reflected also on the test system side, and since the test system depends on the physical machine as well, we introduce this concept for accessing low level information at the level of the test specification.

The `tick` values might be used, for example, to acknowledge how many internal counts passed in between two given instructions, or how many internal counts passed from the initiation of sending a message until the message has actually left the test system.

The time that passes between two clock ticks represent the period of the clock, and it is inversely proportional with the frequency of the CPU. The period and the frequency are properties of the CPU. All the durations between events measured using the clock of the system are divisible by the period of the clock.

The domain associated with this data type will be simply denoted as $\mathbb{T}\mathsf{icks}$.

A.2 Syntax Of Special Operations Relaying On Time: `now`, `wait`, `testcasestart`, `testompontentstart`, `testcomponentstop`

The proposed operations are introduced in the grammar of TTCN-3 as `ControlStatements`, where the non-terminal leaf `RTimeStatements` has been introduced. This is further refined into non-terminals corresponding to each one of the newly introduced statements. The newly introduced statements were already generally described in Section 4.4.2.

From the syntactical construction it can be observed that statement `now` doesn't require parameters; it relies on the internal clock of the system, whose current value is returned every time it is invoked. The value of the clock is returned as a `float` value, representing the number of seconds.

The function of the `wait` statement is to delay the execution of the container thread for a given period of time. It needs, as parameter, the number of seconds for delay as a `float` value and it returns no value.

`testcasestart` operation should return the value of the internal clock when the current test case started. The returned value will be a `float` representing the number of seconds passed since the internal clock was initialized.

`testcomponentstart` and `testcomponentstop` operators should return the value of the internal clock when the associated test component was started, or stopped respectively. The associated test component will be given as a reference in the case of a normal test component, or through the keywords like `mtc`, or `self` for designating the main test component or the current component.

ControlStatement::= TimerStatements | BasicStatements | BehaviourStatements | SUTStatements | StopKeyword | RTimeStatements;

RTStatements::= NowStatement | WaitStatement | TestCaseStartStatement | TestComponentStartStatement | TestComponentStopStatement;

NowStatement::= ***"now"***;

WaitStatement::= ***"wait"*** ***"("*** FloatValue ***")"***;

TestCaseStartStatement::= ***"testcasestart"***;

TestComponentStartStatement::=
[VariableRef | ***"mtc"*** | ***"self"***] ***"."*** ***"testcomponentstart"***;

TestComponentStopStatement::=
[VariableRef | ***"mtc"*** | ***"self"***] ***"."*** ***"testcomponentstop"***;

A.3 Syntax For `receive` With `timestamp`

The syntactic rules for this extension of a receive statement in TTCN-3 are presented below. We introduce the new keyword `timestamp` which is enhancing the `PortRedirect` non-terminal. The `timestamp` token should be followed by a variable identifier associated with a variable that is either of `float` or `timespan` type. This means that the time value that is automatically saved when a new message is received can be saved either as `float` or as `timespan`. This value is saved and added to the global list of variables only if the matching constraints on the port are successfully passed.

ReceiveStatement::=PortOrAny *"."* PortReceiveOp;

PortReceiveOp::=*"receive"* [*"("* ReceiveParameter *")"*][FromClause] [PortRedirect];

PortRedirect::=*"→"*((ValueSpec[SenderSpec] | SenderSpec)[TimeStampSpec]) | TimeStampSpec;

TimeStampSpec::=*"timestamp"*(TimeSpanVarIdentifier | FloatVarIdentifier);

TimeStampVarIdentifier::=Identifier;

FloatVarIdentifier::=Identifier;

Identifier::=Alpha(AlphaNum | Underscore);

Apart from this enhancement of the `PortRedirect` rule, the syntax of the `receive` statement remains unchanged.

A.4 Syntax Of `send` With `timestamp`

The previous section (Section A.3) introduced a mechanism for precisely recording the time value when a message enters into the system. The same necessity, of accurately saving the time value, turns up also for the moment when the message leaves the system. Therefore, we extend the syntax and semantics of `send` instruction accordingly. The syntactic rules for this extension of a `send` statement in TTCN-3 are:

SendStatement::= Port *"."*PortSendOp;

PortSendOp::=*"send""("*SendParameter *")"*[ToClause][TimeStampClause];

TimeStampClause::=*"→"*TimeStampSpec;

The non-terminal `TimeStampClause` is added to the `PortSendOp`. The notation for the `timestamp` mechanism with `send` is similar with the one used for `receive` with `timestamp`. Therefore `TimeStampSpec` non-terminal is reused (see A.3).

A.5 Syntax Of The Temporal Predicates

Temporal predicates are introduced to control the timing of incoming and outgoing messages. A temporal predicate $tp \in \mathbb{T}P$ is an expression that specifies a set of time points. Such a predicate will match a time point $t \in \mathbb{T}S \cup \mathbb{F}$ when the time point is included in the set determined by the predicate $tp \in \mathbb{T}P$.

In general we distinguish between simple temporal predicates and complex temporal predicates. A simple predicate consists of one of the temporal operators at, after, before, and within and their respective negation (e.g. not before). A complex predicate is a combination of predicates that are connected by the logical operators or and and.

However, using the temporal operators we are able to specify ranges of date time values and restrict the availability of TTCN-3 statements with respect to time.

- The at operator takes a time value $t, t \in \mathbb{T}S \cup \mathbb{F}$ and specifies a time constraint according to that value. The negation of the predicate, not at, specifies the domain of time values that are either greater or less than the indicated value.
- The before and after operators are each parameterized with a time value $t, t \in \mathbb{T}S \cup \mathbb{F}$. They specify a range of time points consisting of the values $[0, t]$ in the case of before and $[t, \infty)$ in the case of after. The respective inversions are defined by (t, ∞) and $[0, t)$.
- The within operator is parameterized with two time parameters t_1 and t_2, $(t_1, t_2) \in (\mathbb{T}S \times \mathbb{T}S) \cup (\mathbb{F} \times \mathbb{F})$, that directly denote the boundary of the time interval, thus $[t_1, t_2]$. The negation of the within operator is defined by $[0, t_1)$ and (t_2, ∞).

For the time predicates we introduce completely new rules to the syntax of TTCN-3. For example, `TemporalPredicate` is a new non-terminal used for defining complex temporal predicates. Complex temporal predicates are composed from simple temporal predicates, or from other complex temporal predicates using set conjunctions `and`, `or` as well as the negation `not`.

TemporalPredicate::= TemporalAndPredicate |
TemporalAndPredicate ***"or"***TemporalPredicate;

TemporalAndPredicate::= TemporalSinglePredicate |
TemporalSinglePredicate ***"and"***TemporalAndPredicate;

TemporalSinglePredicate::= [***"not"***](WithinStatement | BeforeAfterAtStatement);

WithinStatement::=*"within""("*TimeIntervalFloat | TimeStampInterval***")"***;

TimeIntervalFloat::=(FloatVarIdentifier | FloatValue)
".."
(FloatVarIdentifier | FloatValue);

TimeStampInterval::=(TimeStampVarIdentifier | TimeStampValue)
".."
(TimeStampVarIdentifier | TimeStampValue);

BeforeAfterAtStatement::=BeforeStatement | AfterStatement | AtStatement;

Simple temporal predicates are of the types `WithinStatement` or `BeforeAfterAtStatement`. As presented in the following, `WithinStatement` predicate requires two time parameters which could be expressed either as `float` values or as `timespan` values. The `BeforeAfterAtStatement` predicates require only one time parameter, expressed also as `float` or `timespan`.

BeforeStatement::=*"before"*
(FloatVarIdentifier | FloatValue | TimeStampVarIdentifier | TimeStampValue);

AfterStatement::=*"after"*
(FloatVarIdentifier | FloatValue | TimeStampVarIdentifier | TimeStampValue);

AtStatement::=*"at"*
(FloatVarIdentifier | FloatValue | TimeStampVarIdentifier | TimeStampValue);

A.6 Syntax For The `receive` Instructions Which Verify Incoming Communication

Applied to receiving TTCN-3 statements (e.g. `receive`, `getcall`, `getreply`, `trigger`, `catch` etc.) the temporal predicates yield as a verification instrument. They are integrated in the TTCN-3 matching mechanism and influence the evaluation of the input queues. That is, a receiving statement with a temporal predicate attached, is only processed successfully when - like in ordinary TTCN-3 - the message value conforms to the value template and - introduced for RT-TTCN-3 - the reception time associated with the message conforms to the temporal predicate. A message conforms to a temporal predicate if, and only if, its time of reception $t_{timestamp}$ is included in the time interval defined by the temporal predicate $t_{template}, t_{template} \in \mathbb{T}P$, where $\mathbb{T}P$ represents the set of all time predicates.

The syntactic extension for the receive statement is given below. The temporal operators can also be applied to `trigger`, `getcall`, `getreply`, and `catch` statements.

ReceiveStatement::=PortOrAny *"."*PortReceiveOp
PortReceiveOp::=*"receive"* [*"("* ReceiveParameter *")"*]
[FromClause][TemporalPredicate] [PortRedirect];

The non-terminal to be enhanced is in this situation `PortReceiveOp`, to which the non-terminal `TemporalPredicate`, that has been defined in the previous section (A.5) is added. The syntax is extended in a consistent manner with previous enhancements. Therefore, the syntax of `receive` operation now incorporates both extensions for time stamping mechanism and for the usage of time predicates.

A.7 Syntax For `send` Instructions Which Control Outgoing Communication

Applied to sending TTCN-3 statements (e.g. `send`, `call`, `reply` etc.) the usage of time predicates for expressing time constraints on the outgoing communication causes the test system to ensure that the statement will be executed in time. Thus, the test system suspends the execution of the component until the first possible temporal match occurs and then continues by dispatching the message or procedure call. Please note, if the system is delayed beyond the accepted time bounds – i.e. no temporal match is possible any more – the test system continues the execution by setting an error verdict. The syntactical extension introduced for adding temporal predicates for all TTCN-3 sending statements are exemplified by the `send` statement. The extensions for `call`, `reply`, and `raise` are defined similar.

The following grammar rules make visible how the `PortSendOp` is enhanced with the `AtStatement` node which is incrementally added to the previously enhanced structure (see Section A.4).

SendStatement::= Port*"."*PortSendOp;
PortSendOp::=*"send""("*SendParameter*")"*
[ToClause][TimeStampClause][AtStatement];

From all the time predicates that were introduced in Section A.5 only `at` statement was considered to be appropriate for expressing constraints on the sending of messages. The other statements would have introduced a time non-determinist on the test system side.

A.8 Syntax For `alt` Instructions Which Control Incoming Communication

As we showed in the previous sections – Sections A.6, A.7 – using temporal predicates we are able to define timing constraints for outgoing communication and also to verify the timings of the incoming messages. The paradigm of imposing time limits for communication on ports can be also applied for statements such as `alt` or `interleave` which might handel incoming communication on more than one port.

The syntactical structure is exemplified by the definition of the `alt` statement, in the following grammar rule:

AltStatement::= ***"alt"***AltGuardList[***"break"***AtStatement][StmtBlock]

The structure of `AltStatement` non-terminal has been enhanced with a continuation sequence introduced by the `break` keyword. This would interrupt the behavior of the `alt` at a certain point in time, if no valid message has been received meanwhile. It provides a meaningful alternative to the blocking behavior of an `alt`, keeping the $\mathcal{TS}$ from being blocked for an unlimited amount of time. The interruption will trigger a handling routine which will acknowledge the time overflow and might also do some additional configurations.

A.9 Syntax For Instructions Controlling The Starting And Stoping Of Test Components

StartTCStatement::=ComponentOrDefaultReference***"." "start"***
"("FunctionInstance***")"***
[TimeStampSpec]
| ComponentOrDefaultReference***"." "start"***
"("FunctionInstance***")"***
[AtStatement]

StopTCStatement::=***"stop"***[TimeStampSpec][AtStatement]
| (ComponentReferenceOrLiteral***"." "stop"***)
[TimeStampSpec][AtStatement]
| ***"all" "component" "." "stop"***
[TimeStampSpec][AtStatement];

Appendix B

Predefined Conversion Functions

This Appendix contains the functions designed to realize the conversions between the different formats proposed for representing time. These formats were introduced as new Real-time TTCN-3 data types in Section 4.4.1 of Chapter 4.

B.1 Converting `timespan` Values To `float`

The conversion function presented in this section is responsible with transforming a value of a time interval, that is represented in `timespan` format, to a `float` value representing the equivalent number of seconds for that time interval.

B.1.1 Syntactic structure:

This syntax is provided for integrating the function into the grammar of TTCN-3.

timespan2seconds(TimeSpanValue)

B.1.2 Signature:

```
float timespan2seconds(timespan p_timespan);
```

The signature of the function is presented here. As displayed, the function takes a `timespan` value as an input parameter and on its basis, it returns a `float` value. The `float` value represents the equivalent number of seconds that were computed on the basis of the input parameter.

B.1.3 Semantic description:

The function returns a `float` value, which represents the equivalent in seconds of the `timespan` value given as parameter. It is a conversion function whose computing logic is synthesized in Rule B.1.

$$\frac{\begin{array}{c} ts = a * hour + b * min + c * sec + d * milisec + e * microsec + f * nanosec, \\ ts \in \mathbb{TS}, \{a, b, c, d, e, f\} \in \mathbb{F} \end{array}}{\begin{array}{c} timespan2seconds(ts) \rightarrow \\ (a * 60 * 60 + b * 60 + c + d * 10^{-3} + e * 10^{-6} + f * 10^{-9}) \in \mathbb{F} \end{array}} \tag{B.1}$$

B.2 Converting Seconds To A `timespan` Value

The conversion function presented in this section is responsible with transforming a value of a time interval, that is represented in number of seconds as `float`, to an equivalent representation as `timespan` value. This function is complementary to the one from the previous section (Section B.1).

B.2.1 Syntactic structure:

This syntax is provided for integrating the function into the grammar of TTCN-3.

seconds2timespan(FloatValue)

B.2.2 Signature:

```
timespan seconds2timespan(float p_number_of_seconds);
```

The signature of the function is presented here. As it shows, the function takes a `float` value as an input parameter and on its basis, it returns a `timespan` value. The `timespan` is computed on the basis of the input parameter as indicated in Rule B.2.

B.2.3 Semantic description:

This function should return a `timespan` value which has been created on the basis of the number of seconds provided as a `float` parameter. This function is introduced for compatibility purposes and for the easy manipulation of time values. It is a conversion function whose computing logic is synthesized in Rule B.2.

$$\frac{s \in \mathbb{F}}{seconds2timespan(s) \rightarrow (s * sec) \in \mathbb{TS}} \tag{B.2}$$

B.3 Converting `tick` Values To Seconds

The conversion function presented in this section is responsible for transforming a value of a time interval, that is represented in clock `tick` format, to a `float` value representing the equivalent number of seconds for that time interval.

B.3.1 Syntactic structure:

This syntax is provided for integrating the function into the grammar of TTCN-3.

ticks2seconds(TickValue)

B.3.2 Signature:

```
float ticks2seconds(tick p_number_of_ticks);
```

The signature of the function is presented here. As displayed, the function takes a clock `tick` value as an input parameter and on its basis, it returns a `float` value. The `float` value represents the equivalent number of seconds that were computed on the basis of the input parameter.

B.3.3 Semantic description:

The returned value will be a `float` number indicating how many seconds are associated with the given number of clock ticks. If the frequency f of the internal clock of the system is given as ticks per second, the calculation is performed as following in Rule B.3:

$$\frac{\delta t = \frac{1}{f} \in \mathbb{F}, ticks \in \mathsf{Tick}}{ticks2seconds(ticks) \rightarrow \delta t * ticks} \tag{B.3}$$

B.4 Converting Seconds To A `tick` Value

The conversion function presented in this section is responsible for transforming a value of a time interval, that is represented in number of seconds as `float`, to an equivalent representation as clock `tick` value. This function is complementary to the one from the previous section (Section B.3).

B.4.1 Syntactic structure:

This syntax is provided for integrating the function into the grammar of TTCN-3.

seconds2ticks(FloatValue)

B.4.2 Signature:

```
tick seconds2ticks(float p_number_of_seconds);
```

The signature of the function is presented here. As displayed, the function takes a `float` value as an input parameter and on its basis, it returns a clock `tick` value. The clock `tick` is computed on the basis of the input parameter as indicated in Rule B.4.

B.4.3 Semantic description:

The function returns a natural number value which represents the number of processor ticks that passed in the time interval given as parameter. The parameter is a `float` value representing the interval of time in seconds. Based on the frequency of the clock, the function evaluation rule is expressed in Rule B.4.

$$\frac{\delta t = \frac{1}{f}, s \in \mathbb{F}}{seconds2ticks(s) \rightarrow \lfloor \frac{s}{\delta t} \rfloor} \tag{B.4}$$

In order to complete semantic specification from Subsection C, the sematic for the evaluation of the comparison expressions with `timespan` values is presented in the following:

$$\frac{\begin{array}{c} timespan2seconds(ts_1) BOP timespan2seconds(ts_2) \rightarrow true, \\ BOP \in CompOP \end{array}}{ts_1 BOP ts_2 \rightarrow true} \tag{B.5}$$

$$\frac{\begin{array}{c} timespan2seconds(ts_1) BOP timespan2seconds(ts_2) \rightarrow false, \\ BOP \in CompOP \end{array}}{ts_1 BOP ts_2 \rightarrow false} \tag{B.6}$$

Appendix C

Time Expressions With Numerical And Logical Operators

This Appendix contains the semantics for time expressions with numerical and logical operators. The time expressions are evaluated to values of different time data types. These time data types were introduced as new Real-time TTCN-3 data types in Section 4.4.1 of Chapter 4.

Let us consider the following domains for the data types previously introduced: $\mathbb{TS}$ for the `timespan` values, $\mathbb{F}$ for floats, and $\mathbb{T}\mathsf{icks}$ for `tick` values. Additionally, we consider $\mathbb{N}^{>0}$ for natural strict positive numbers, and $\mathbb{B}\mathsf{ool}$ for boolean set $\{true, false\}$.

In the following we present the semantics for typing expressions that are built with operands from these domains. The operators for building the expressions are the binary mathematical and comparison operators: $NumOP = \{+, -, *, /\}$, $CompOP = \{<, >, \geq, \leq, ==\}$, where the domains on which the numerical and comparison operators are applied are as follows:

$+ : \mathbb{TS} \times \mathbb{TS} \to \mathbb{TS}$, or in infix notation : $\mathbb{TS} + \mathbb{TS} \to \mathbb{TS}$
$- : \mathbb{TS} \times \mathbb{TS} \to \mathbb{TS}$, or in infix notation : $\mathbb{TS} + \mathbb{TS} \to \mathbb{TS}$
$* : (\mathbb{F} \bigcup \mathbb{N}) \times \mathbb{TS} \to \mathbb{TS}$, or in infix notation : $(\mathbb{F} \bigcup \mathbb{N}) * \mathbb{TS} \to \mathbb{TS}$
$/ : \mathbb{TS} \times (\mathbb{F}^* \bigcup \mathbb{N}^*) \to \mathbb{TS}$, or in infix notation : $\mathbb{TS}/(\mathbb{F}^* \bigcup \mathbb{N}^*) \to \mathbb{TS}$
$<: \mathbb{TS} \times \mathbb{TS} \to \mathbb{B}\mathsf{ool}$, or in infix notation : $\mathbb{TS} < \mathbb{TS} \to \mathbb{B}\mathsf{ool}$
$>: \mathbb{TS} \times \mathbb{TS} \to \mathbb{B}\mathsf{ool}$, or in infix notation : $\mathbb{TS} > \mathbb{TS} \to \mathbb{B}\mathsf{ool}$
$\leq: \mathbb{TS} \times \mathbb{TS} \to \mathbb{B}\mathsf{ool}$, or in infix notation : $\mathbb{TS} \leq \mathbb{TS} \to \mathbb{B}\mathsf{ool}$
$\geq: \mathbb{TS} \times \mathbb{TS} \to \mathbb{B}\mathsf{ool}$, or in infix notation : $\mathbb{TS} \geq \mathbb{TS} \to \mathbb{B}\mathsf{ool}$
$==: \mathbb{TS} \times \mathbb{TS} \to \mathbb{B}\mathsf{ool}$, or in infix notation : $\mathbb{TS} == \mathbb{TS} \to \mathbb{B}\mathsf{ool}$

The semantics for typing the expressions is going to be build in the following manner:

The assertion that if $S_1, S_2, S_3 ... S_n$, all of which evaluates to *true*, implies statements S evaluates to *true*, is represented by $\frac{S_1, S_2, S_3 ... S_n}{S}$.

If e is an expression and v is a value, then $e \to v$ means that expression e evaluates to value v; if e and e' are two given expressions, then $e \to e'$ means that expression e evaluates to e'. If $f(t)$ has a one variable function and $e(t)$ is one expression in which variable t is not bound to a fixed value, then $f(t) \to e(t)$ means that $f(t)$ evaluates to

$e(t)$. We consider as an implicit assumption that $v \to v$, that means that v evaluates always to v, where v is a value.

Based on these conventions, we define the type semantics of expressions trough the following rules:

In Rule C.1 there are two `timespan` values in the canonical format, and the result of the addition is also a `timespan` value, presented also in the canonical format.

$$\frac{\begin{array}{c} ts_1, ts_2 \in \mathbb{TS} \\ ts_1 = h_1 * hour + m_1 * min + s_1 * sec + ms_1 * milisec + \mu s_1 * microsec + ns_1 * nanosec, \\ ts_2 = h_2 * hour + m_2 * min + s_2 * sec + ms_2 * milisec + \mu s_2 * microsec + ns_2 * nanosec \end{array}}{\begin{array}{c} ts_1 \pm ts_2 \in \mathbb{TS} \wedge ts_1 \pm ts_2 = (h_1 \pm h_2) * hour + (m_1 \pm m_2) * min + \\ (s_1 \pm s_2) * sec + (ms_1 \pm ms_2) * milisec + \\ (\mu s_1 \pm \mu s_2) * microsec + (ns_1 \pm ns_2) * nanosec \end{array}} \tag{C.1}$$

Rules C.2, C.3 show how an expression which evaluates to a `timespan` value should be also typed as `timespan`, and the additive composition of two such expressions evaluate to a `timespan` value as well. Rules C.4, and C.5 show how multiplication with a `float` or `natural` value affect the type of the formed expression. Furthermore, Rules C.6, C.7 state that the expression formed by comparing two `timespan` values should be typed as `boolean`, and the same will happen with the comparison of two expressions of type `timestamp`.

$$\frac{e \to ts, ts \in \mathbb{TS}}{e \in \mathbb{TS}} \tag{C.2}$$

$$\frac{e_1 \to ts_1, e_2 \to ts_2, ts_1, ts_2 \in \mathbb{TS}}{e_1 \pm e_2 \to ts_1 \pm ts_2} \tag{C.3}$$

$$\frac{x \in \mathbb{F} \cup \mathbb{N}^{>0}, ts \in \mathbb{TS}}{x * ts \in \mathbb{TS}} \tag{C.4}$$

$$\frac{e \to ts, x \in \mathbb{F}, ts \in \mathbb{TS}}{x * e \to x * ts} \tag{C.5}$$

$$\frac{ts_1, ts_2 \in \mathbb{TS}, BOP \in CompOP}{ts_1 BOP ts_2 \in \mathbb{B}\mathsf{ool}} \tag{C.6}$$

$$\frac{e_1 \to ts_1, e_2 \to ts_2}{e_1 BOP e_2 \to ts_1 BOP ts_2} \tag{C.7}$$

In the following, the same rules(C.8 – C.14) apply also to the `tick` values, which are basically `natural` numbers, which makes the typing semantic to be trivial.

$$\frac{tick_1, tick_2 \in \mathbb{T}\mathsf{icks}}{tick_1 \pm tick_2 = tick' \in \mathbb{T}\mathsf{icks}} \tag{C.8}$$

$$\frac{e \to tick, tick \in \mathbb{T}\mathsf{icks}}{e \in \mathbb{T}\mathsf{icks}} \tag{C.9}$$

$$\frac{e_1 \to tick_1, e_2 \to tick_2, tick_1, tick_2 \in \mathbb{T}\mathsf{icks}}{e_1 \pm e_2 \to tick_2 \pm tick_2} \tag{C.10}$$

$$\frac{x \in \mathbb{F} \cup \mathbb{N}^{>0}, tick \in \mathbb{T}\mathsf{ick}}{x * tick \in \mathbb{T}\mathsf{icks}} \quad \text{(C.11)}$$

$$\frac{e \to tick, tick \in \mathbb{T}\mathsf{icks}, x \in \mathbb{F} \cup \mathbb{N}^{>0}}{x * e \to x * tick} \quad \text{(C.12)}$$

$$\frac{tick_1, tick_2 \in \mathbb{T}\mathsf{icks}, BOP \in CompOP}{tick_1 BOP tick_2 \in \mathbb{B}\mathsf{ool}} \quad \text{(C.13)}$$

$$\frac{e_1 \to tick_1, e_2 \to tick_2}{e_1 BOP e_2 \to tick_1 BOP tick_2} \quad \text{(C.14)}$$

Some predefined conversion functions for translating the data from one format to another are provided in Appendix B.

Appendix D

Semantics Completions Of The Real-time Extensions Of TTCN-3 Using Logic Rules

This Appendix contains the complementary logic rules to the semantics of the new real-time extensions for TTCN-3, which has been presented in section 4.5, Chapter 4.

The conventions used for expressing the semantics of the automata through logic rules takes the following form: $instruction : \frac{antecedent(s)}{conclusion}[side\ condition]$

D.1 Semantics Of Special Operations Relaying On Time: `now`, `wait`, `testcasestart`, `testcomponentstart`, `testcomponentstop`

The semantic rules expressing the functionality of the `clock` and `now` automata are presented below. Rule D.1 states that the edge from state S_{clock} to itself is transited when the guard $c_{clock} == \delta t$ is satisfied. During the transition, the value of c_{clock} is also updated. Rules D.2, D.3 state that the edges from state S_{now} to itself are transited when the signals $tick!$, and respectively $now!$ are received. During the transitions, the values of c_0 and v_{now} are updated accordingly.

$$\mathsf{clock} : \frac{S_{clock} \xrightarrow{c_{clock}==\delta t, tick!, c_{clock}:=0} S_{clock}}{if\ (c_{clock} == \delta t)\ then\ S_{clock} \rightarrow tick!S_{clock} \wedge c_{clock} := 0}[c_{clock} \in \mathcal{C}\mathsf{locks}] \tag{D.1}$$

$$\mathsf{now} : \frac{S_{now} \xrightarrow{tick?, c_0 := c_0 + \delta t} S_{now}}{tick!S_{now} \rightarrow S_{now} \wedge c_0 := c_0 + \delta t}[c_0 \in \mathcal{C}\mathsf{locks}] \tag{D.2}$$

$$\mathsf{now} : \frac{S_{now} \xrightarrow{now?, v_{now} := c_0} S_{now}}{now!S_{now} \rightarrow S_{now} \wedge v_{now} := c_0}[v_{now} \in \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}] \tag{D.3}$$

The result of the composition of the two automata might be expressed through rule D.4. The transition of the composed automata is triggered by the satisfaction of guard $c_{clock} == \delta t$. During the transition, the values of the two clocks - c_o and c_{clock} - are updated.

$$\mathsf{now, clock} : \frac{(S_{clock} \wedge S_{now}) \wedge c_{clock} == \delta t}{(S_{clock} \wedge S_{now}) \rightarrow (S_{clock} \wedge S_{now}) \wedge c_0 := c_0 + \delta t \wedge c_{clock} := 0}[c_0, c_{clock} \in \mathcal{C}\mathsf{locks}] \tag{D.4}$$

Rules D.5, D.6 D.7 express the semantics of transitions on the three edges of the `wait` automata. It should be emphasized that the `wait` automata synchronizes with the `clock` automata through the tick signal.

$$\mathsf{wait}: \frac{S_{wait} \xrightarrow{tick?, c_{wait} := c_{wait} + \delta t} S_{wait}}{tick! S_{wait} \rightarrow S_{wait} \wedge c_{wait} := c_{wait} + \delta t}[c_{wait} \in \mathcal{C}\mathsf{locks}] \tag{D.5}$$

$$\mathsf{wait}: \frac{S_{wait} \xrightarrow{(u-\delta\epsilon) \leq c_{wait} \leq (u+\delta\epsilon)} S_{next}}{if\ ((u-\delta\epsilon) \leq c_{wait} \leq (u+\delta\epsilon))\ then\ S_{wait} \rightarrow S_{next}}[c_{wait} \in \mathcal{C}\mathsf{locks}, u \in \mathbb{R}^{+}] \tag{D.6}$$

$$\mathsf{wait}: \frac{S_{wait} \xrightarrow{c_{wait} > (u+\delta\epsilon)} S_{error}}{if\ (c_{wait} > (u+\delta\epsilon))\ then\ S_{wait} \rightarrow S_{error}}[c_{wait} \in \mathcal{C}\mathsf{locks}, u \in \mathbb{R}^{+}] \tag{D.7}$$

$$\mathsf{testcasestart}: \frac{S_{tc\_start} \xrightarrow{now!, v_{testcasestart} := v_{now}} S_{next}}{S_{tc\_start} \rightarrow now! S_{next} \wedge v_{testcasestart} := v_{now}}[v_{testcasestart}, v_{now} \in \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}] \tag{D.8}$$

In Rule notation the semantics of the `testcasestart`, `testcomponentstart` and `testcomponentstop` is presented in(D.8, D.9 and D.10).

testcomponentstart :

$$\frac{S^{i}_{comp\_start} \xrightarrow{now!, v^{i}_{testcomponentstart} := v_{now}} S_{next}}{S_{comp\_start} \rightarrow now! S_{next} \wedge v^{i}_{testcomponentstart} := v_{now}}[v^{i}_{testcomponentstart}, v_{now} \in \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}] \tag{D.9}$$

testcomponentstop :

$$\frac{S^{i}_{comp\_stop} \xrightarrow{now!, v^{i}_{testcomponentstop} := v_{now}} S_{next}}{S_{comp\_stop} \rightarrow now! S_{next} \wedge v^{i}_{testcomponentstop} := v_{now}}[v^{i}_{testcomponentstop}, v_{now} \in \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}] \tag{D.10}$$

D.2 Semantics For `receive` With `timestamp`

The transition branches of the state automata from Figure 4.9 and from Figure 4.10 are expanded into the set of Rules D.11, D.12 and D.13, D.14, D.15 accordingly.

Rule D.11 states that if a new message is received the timed automata transits from the waiting state into a service state when the current time is asked through the use of the *now*! signal and the value is saved in a time stamp associated with the received message.

Both the message and the time stamp are enqueued in a queue associated with the port on which the message arrived. The Rule D.12 continues the input handling routine by sending a wake up message to the timed automata responsible with the checking of the message and after that it returns to the waiting state.

receive :

$$\frac{S_{receive} \xrightarrow{ch_i(e?),msg_{ij}:=input(ch_i)} S^i_{rcv\_timestamp} \;\wedge\; S^i_{rcv\_timestamp} \xrightarrow{now!,timestamp_{ij}:=v_{now},queue_i.enqueue(msg_{ij},timestamp_{ij})} S^i_{start\_match}}{ch_i(e!)S_{receive} \to now!S^i_{start\_match} \;\wedge\; msg_{ij} = input(ch_i) \;\wedge\; timestamp_{ij} = v_{now} \;\wedge\; queue_i.enqueue(msg_{ij}, timestamp_{ij})} \tag{D.11}$$

receive :

$$\frac{S^i_{start\_match} \xrightarrow{queue_i!} S_{receive}}{S^i_{start\_match} \to queue_i!S_{receive}} [queue_i! \in \mathcal{A}'] \tag{D.12}$$

Rules D.13-D.15 describe the logic of the `match` automata: when the wake up signal is received it takes the message from the associated queue and tries to match it against the given template; if the match fails, it returns to the waiting state, and if the match succeeds it saves both message and timestamp into variables which are then added to the $\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}$, not before sending the *received*! signal to inform the $\mathcal{TS}$ that the received operation succeeded and it should move forward.

match :

$$\frac{S^i_{match\_wait} \xrightarrow{queue_i?} S^{i1}_{match}}{queue_i!S^i_{match\_wait} \to S^{i1}_{match} \;\wedge\; (msg_{ij}, timestamp_{ij}) := queue_i.dequeue()} \tag{D.13}$$

match :

$$\frac{S^{i1}_{match} \;\wedge\; msg_{ij} \in Tmpl_{i1}}{S^{i1}_{match} \to received!S^i_{match\_stop} \;\wedge\; \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} := \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} \bigcup \{msg_{ij}, timestamp_{ij}\}} \tag{D.14}$$

match :

$$\frac{S^{i1}_{match} \;\wedge\; msg_{ij} \notin Tmpl_{i1}}{S^{i1}_{match} \to S^i_{match\_wait}} \tag{D.15}$$

D.3 Semantics Of `send` With `timestamp`

Rules D.16, D.17 expand the logic of the transitions in the `send` state automata. From state S_{send} to state $S^i_{snd\_timestamp}$ the output event is triggered. On the transition between $S^i_{snd\_timestamp}$ current time is requested and saved to a global variable. Time request is addressed to the `now` automata (see A.2) which is running in the background.

send :

$$\frac{S_{send} \xrightarrow{ch_i(e!)} S^i_{snd\_timestamp}}{S_{send} \rightarrow ch_i(e!) S^i_{snd\_timestamp}} [ch_i(e!) \in Ch_i(\mathcal{A}_{out})] \tag{D.16}$$

send :

$$\frac{S^i_{snd\_timestamp} \xrightarrow{now!, s\_tmestamp := v_{now}} S_{next}}{S_{send} \rightarrow S_{next} \ \wedge \ s\_timestamp := v_{now} \ \wedge \ \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} := \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} \cup \{s\_timestamp_{ij}\}} \tag{D.17}$$

D.4 Semantics Of The Temporal Predicates

The semantics for evaluating temporal predicates is going to be build in the following manner:

The assertion that if $S_1, S_2, S_3 ... S_n$, all of which evaluates to *true*, implies statements S evaluates to *true*, is represented by $\frac{S_1, S_2, S_3 ... S_n}{S}$. If e is an expression and v is a value, then $e \rightarrow v$ means that expression e evaluates to value v.

Rules D.18, D.20, D.22, D.24, D.26, D.27 are validity rules, stating that there are values of the requested types that could fulfill the predicates, or that the predicates are valid. Rules D.19, D.21, D.23, D.25 describe how it is established whether or not a given value fulfills the indicated predicate.

$$within(tp_1, tp_2) : \frac{tp_1, tp_2 \in \mathbb{T}P}{\exists t \in \mathbb{T}P \wedge tp_2 \leq t \leq tp_2} \tag{D.18}$$

$$within_{t_0}(tp_1, tp_2) : \frac{t_0, tp_1, tp_2 \in \mathbb{T}P}{\begin{array}{c} if \ (tp_1 \leq t_0 \leq tp_2) \\ then \ within_{t_0}(tp_1, tp_2) \rightarrow true \\ else \ within_{t_0}(tp_1, tp_2) \rightarrow false \end{array}} \tag{D.19}$$

$$before \ tp : \frac{tp \in TP}{\exists t \in \mathbb{T}P \wedge tp \leq t} \tag{D.20}$$

$$before_{t_0} \ tp : \frac{t_0, tp \in \mathbb{T}P}{\begin{array}{c} if \ (t_0 \leq tp) \\ then \ before_{t_0} \ tp \rightarrow true \\ else \ before_{t_0} \ tp \rightarrow false \end{array}} \tag{D.21}$$

$$after \ tp : \frac{tp \in \mathbb{T}P}{\exists t \in \mathbb{T}P \wedge tp \geq t} \tag{D.22}$$

$$after_{t_0}\ tp : \frac{t_0, tp \in \mathbb{T}P}{\begin{array}{c} if\ (t_0 \geq tp) \\ then\ after_{t_0}\ tp \rightarrow true \\ else\ after_{t_0}\ tp \rightarrow false \end{array}} \tag{D.23}$$

$$at\ tp : \frac{tp \in \mathbb{T}P}{\exists t \in \mathbb{T}P \wedge tp == t} \tag{D.24}$$

$$at_{t_0}\ tp : \frac{t_0, tp \in \mathbb{T}P}{\begin{array}{c} if\ (t_0 == tp) \\ then\ at_{t_0}\ tp \rightarrow true \\ else\ at_{t_0}\ tp \rightarrow false \end{array}} \tag{D.25}$$

$$TP_1\ or\ TP_2 : \frac{TP_1, TP_2 \subset \mathbb{T}P}{\exists t \in \mathbb{T}P \wedge (t \in TP_1 \vee t \in TP_2)} \tag{D.26}$$

$$TP_1\ and\ TP_2 : \frac{TP_1, TP_2 \subset \mathbb{T}P}{\exists t \in \mathbb{T}P \wedge (t \in TP_1 \wedge t \in TP_2)} \tag{D.27}$$

D.5 Semantics For The `receive` Instructions Which Verify Incoming Communication

Rules D.28, D.29 which describes the transitions for the `receive with time predicate` automata are identical with the ones for the `receive with timestamp` automata. Newly introduced in this context, is Rule D.30, indicating a transition to an error state.

receive with time predicate :

$$\frac{\begin{array}{c} S_{receive} \xrightarrow{ch_i(e?), msg_{ij} := input(ch_i)} S^i_{rcv\_timestamp} \wedge \\ S^i_{rcv\_timestamp} \xrightarrow{now!, timestamp_{ij} := v_{now}, queue_i.enqueue(msg_{ij}, timestamp_{ij})} S^i_{start\_match} \end{array}}{\begin{array}{c} ch_i(e!) S_{receive} \rightarrow now! S^i_{start\_match} \wedge \\ msg_{ij} = input(ch_i) \ \wedge \ timestamp_{ij} = v_{now} \wedge \\ queue_i.enqueue(msg_{ij}, timestamp_{ij}) \end{array}} \tag{D.28}$$

receive with time predicate :

$$\frac{S^i_{start\_match} \xrightarrow{queue_i!} S_{receive}}{S^i_{start\_match} \rightarrow queue_i! S_{receive}} [queue_i! \in \mathcal{A}'] \tag{D.29}$$

receive with time predicate :

$$\frac{S_{receive} \xrightarrow{stop\_receive?} S_{error}}{stop\_receive!S_{receive} \rightarrow S_{error}} \tag{D.30}$$

The rules for describing the transitions of the `match` automata are presented below. After the automata is woken up by the $queue_i$ signal in Rule D.31, the automata enters a state where the time constraints are to be matched. These are verified based on the time stamp that was taken and saved when the message arrived, $timestamp_{ij}$. If this verification is passed, the transition for the next match, the structural match state, is taken (Rule D.32). If the time predicate has not yet been satisfied, but is still a valid predicate - there are chances left for a future incoming message to satisfy it - then the automata transits back to the waiting state in Rule D.33. If the time predicate was not yet satisfied and the predicate becomes invalid - the time frame expired - then the `match` automata signalizes error to the `receive` and enters a terminal error state in Rule D.34. If the time predicate was satisfied after state $S^{i1}_{match\_time}$ then the transition is taken to the structural matching state. If the structural matching is passed, then the $received!$ signal is emitted to the `receive` automata and the `receive with time constraints` operation will be successfully accomplished in Rule D.35. $S^{i}_{match\_stop}$ represents a terminal state as well. This state indicates a successful termination of the process. If the structural match is not passed, the `match` automata transits back to the initial waiting state in Rule D.36.

match :

$$\frac{S^{i}_{match\_wait} \xrightarrow{queue_i?} S^{i1}_{match\_time}}{queue_i!S^{i}_{match\_wait} \rightarrow S^{i1}_{match\_time} \ \wedge \ (msg_{ij}, timestamp_{ij}) := queue_i.dequeue()} \tag{D.31}$$

match :

$$\frac{S^{i1}_{match\_time} \xrightarrow{timestamp_{ij} \in \mathbb{T}P} S^{i1}_{match} \ \wedge \ timestamp_{ij} \in TP_{i1}}{S^{i1}_{match\_time} \rightarrow S^{i1}_{match}} \tag{D.32}$$

match :

$$\frac{S^{i1}_{match\_time} \xrightarrow{timestamp_{ij} \notin TP_{i1}} S_{error} \ \wedge \\ timestamp_{ij} \notin TP_{i1} \ \wedge (\exists t \in \mathbb{R}^{+}, t > timestamp_{ij} \ \wedge \ t \in TP_{i1})}{S^{i1}_{match\_time} \rightarrow S^{i}_{match\_wait}} \tag{D.33}$$

match :

$$\frac{S^{i1}_{match\_time} \xrightarrow{timestamp_{ij} \notin TP_{i1}} S_{error} \ \wedge \\ timestamp_{ij} \notin TP_{i1} \ \wedge (\nexists t \in \mathbb{R}^{+}, t > timestamp_{ij} \ \wedge \ t \in TP_{i1})}{S^{i1}_{match\_time} \rightarrow S^{i}_{match\_wait}} \tag{D.34}$$

match :

$$\frac{S^{i1}_{match} \ \wedge \ msg_{ij} \in Tmpl_{i1}}{S^{i1}_{match} \rightarrow received!S^{i}_{match\_stop} \ \wedge \ \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} := \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} \bigcup \{msg_{ij}, timestamp_{ij}\}} \tag{D.35}$$

match :

$$\frac{S^{i1}_{match} \ \wedge \ msg_{ij} \notin Tmpl_{i1}}{S^{i1}_{match} \rightarrow S^{i}_{match\_wait}} \tag{D.36}$$

D.6 Semantics For `send` Instructions Which Control Outgoing Communication

The Rules D.37, D.38 express the semantics of the transitions for the above presented timed automata. At every clock tick the current time value is interrogated. If the current time value is greater than the upper tolerated time limit, then the `send` operation could not be executed as scheduled, and an error routine is triggered.

send at :

$$\frac{S_{send} \xrightarrow{tick?, now!, t_{max}-\delta\epsilon \leq v_{now} \leq t_{max}+\delta\epsilon, ch_i(e!)} S_{next} \ \wedge \ t_{max} - \delta\epsilon \leq (now!v_{now}) \leq t_{max} + \delta\epsilon}{tick!S_{send} \rightarrow ch_i(e!)S_{next}} \tag{D.37}$$

send at :

$$\frac{S_{send} \xrightarrow{tick?, now!, v_{now} > t_{max}+\delta\epsilon} S_{error} \ \wedge \ (now!v_{now}) > t_{max} + \delta\epsilon}{tick!S_{send} \rightarrow S_{error}} \tag{D.38}$$

D.7 Semantics for `alt` Instructions which Control Incoming Communication

The functionality of the `alt` automata is captured also by Rules D.39-D.43.

Once received and time-stamped, the message is passed to the extended `match` automata, to be verified whether or not it respects the temporal and structural constraints associated with that port. The `match` automata defined here represents a generalization for the automata presented in Figure 4.13. In an `alt` statement one port might be used in more than one `receive` branches. On each of the `receive` branches there might be temporal and structural constraints expressed as time predicates and message templates. We consider one `match` automata to be responsible for verifying all the constraints associated with the port correspondent to that automata. The matching will be performed in order in which the `receive` branch has been encountered inside the `alt` statement, from top to the bottom. The first (l^{th}) matching state of the couples – time constraint (TP_{ij}), structural constraint ($Tmpl_{ij}$) – this is satisfied by the received message triggering the $receive_{il}!$ signal which indicates to the `alt` automata that one of the branches has been satisfied. This becomes a success scenario when the `alt` is satisfied, the extended `match` automata reaches a successful terminal state and the execution of the test system moves to the next state. This functionality is expressed by the Rules D.44-D.51.

If the match is not successful, despite trying all the constraints associated with the port, the extended `match` automata associated with that port moves into a waiting state. If none of the branches of the `alt` statement were satisfied in the required amount of time indicated by the parameter of the `alt` statement, then the execution of the `alt` automata is interrupted by a signal given by the `wait_alt` automata. This signalize that an alternative error behavior should be run in this situation. This alternative behavior is the one associated with the `break` instruction. Rules D.44, D.45 are used for describing this behavior.

alt :

$$\frac{S_{alt} \xrightarrow{ch_i(e?),msg_{ij}:=input(ch_i)} S^i_{rcv\_timestamp}}{ch_i(e!)S_{alt} \to S^i_{rcv\_timestamp} \ \wedge \ msg_{ij} := input(ch_i)} \tag{D.39}$$

alt :

$$\frac{S^i_{rcv\_timestamp} \xrightarrow{now!,timestamp_{ij}:=v_{now},queue_i.enqueue(msg_{ij},timestamp_{ij})} S^i_{start\_match}}{\begin{array}{c}S^i_{rcv\_timestamp} \to S^i_{start\_match} \ \wedge \ timestamp_{ij} := v_{now} \ \wedge \\ queue_i.enqueue(msg_{ij}, timestamp_{ij})\end{array}} \tag{D.40}$$

alt :

$$\frac{S^i_{start\_match} \xrightarrow{queue_i!} S_{alt}}{S^i_{start\_match} \to queue_i!S_{alt}} \tag{D.41}$$

alt :

$$\frac{S_{alt} \xrightarrow{received_{il}?} S^j_{brunch} \ \wedge \ S^j_{brunch} \to S_{next}}{received_{il}!S_{alt} \to S_{next}} [\mathcal{B}(queue_i, TP_{il}, Tmpl_{il}) = branch_j] \tag{D.42}$$

alt :

$$\frac{S_{alt} \xrightarrow{break?} S_{break}}{break!S_{alt} \to S_{break}} \tag{D.43}$$

wait_alt :

$$\frac{S_{wait\_alt} \xrightarrow{tick?,now!,v_{now}<t_{max}+\delta\epsilon} S_{wait\_alt} \ \wedge \ now!v_{now} < t_{max} + \delta\epsilon}{tick!S_{wait\_alt} \to S_{wait\_alt}} \tag{D.44}$$

wait_alt :

$$\frac{S_{wait\_alt} \xrightarrow{tick?,now!,v_{now}\geq t_{max}+\delta\epsilon,break!} S_{alt\_stop} \ \wedge \ now!v_{now} \geq t_{max} + \delta\epsilon}{tick!S_{wait\_alt} \to break!S_{alt\_stop}} \tag{D.45}$$

match_alt :

$$\frac{S^{i}_{match\_wait} \xrightarrow{queue_i?,(msg_{ij},timestamp_{ij}):=queue_i.dequeue()} S^{i}_{match}}{queue_i!S^{i}_{match\_wait} \rightarrow S^{ij}_{match} \ \wedge \ (msg_{ij}, timestamp_{ij}) := queue_i.dequeue()} \tag{D.46}$$

match_alt :

$$\frac{S^{i1}_{match} \xrightarrow{msg_{ij}\in Tmpl_{i1} \ \wedge \ timestamp_{ij}\in TP_{i1},received_{i1}!,\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}:=\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}\bigcup\{msg_{ij},timestamp_{ij}\}} S^{i}_{match\_stop} \ \wedge \\ msg_{ij} \in Tmpl_{i1} \ \wedge \ timestamp_{ij} \in TP_{i1}}{S^{i1}_{match} \rightarrow received_{i1}S^{i}_{match\_stop} \ \wedge \\ \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} := \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}\bigcup\{msg_{ij}, timestamp_{ij}\}} \tag{D.47}$$

match_alt :

$$\frac{S^{i1}_{match} \xrightarrow{msg_{ij}\notin Tmpl_{i1} \ \vee \ timestamp_{ij}\notin TP_{i1}} S^{i2}_{match} \ \wedge \\ msg_{ij} \notin Tmpl_{i1} \ \vee \ timestamp_{ij} \notin TP_{i1}}{S^{i1}_{match} \rightarrow S^{i2}_{match}} \tag{D.48}$$

match_alt :

$$\frac{S^{i2}_{match} \xrightarrow{msg_{ij}\in Tmpl_{i2} \ \wedge \ timestamp_{ij}\in TP_{i2},received_{i2}!,\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}:=\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}\bigcup\{msg_{ij},timestamp_{ij}\}} S^{i}_{match\_stop} \ \wedge \\ msg_{ij} \in Tmpl_{i2} \ \wedge \ timestamp_{ij} \in TP_{i2}}{S^{i2}_{match} \rightarrow received_{i2}S^{i}_{match\_stop} \ \wedge \\ \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} := \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}\bigcup\{msg_{ij}, timestamp_{ij}\}} \tag{D.49}$$

...

match_alt :

$$\frac{S^{ib_i}_{match} \xrightarrow{msg_{ij}\in Tmpl_{ib_i} \ \wedge \ timestamp_{ij}\in TP_{ib_i},received_{ib_i}!,\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}:=\mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}\bigcup\{msg_{ij},timestamp_{ij}\}} S^{i}_{match\_stop} \ \wedge \\ msg_{ij} \in Tmpl_{ib_i} \ \wedge \ timestamp_{ij} \in TP_{ib_i}}{S^{ib_i}_{match} \rightarrow received_{ib_i}S^{i}_{match\_stop} \ \wedge \\ \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist} := \mathcal{V}\mathsf{ar}\mathcal{L}\mathsf{ist}\bigcup\{msg_{ij}, timestamp_{ij}\}} \tag{D.50}$$

match_alt :

$$\frac{S^{ib_i}_{match} \xrightarrow{msg_{ij}\notin Tmpl_{ib_i} \ \vee \ timestamp_{ij}\notin TP_{ib_i}} S^{i}_{match\_wait} \ \wedge \\ msg_{ij} \notin Tmpl_{ib_i} \ \vee \ timestamp_{ij} \notin TP_{ib_i}}{S^{ib_i}_{match} \rightarrow S^{i}_{match\_wait}} \tag{D.51}$$

D.8 Semantics For Instructions Controlling The Starting And Stoping Of Test Components

start_component at :

$$\frac{S_{start\_comp} \xrightarrow{tick?, now!, t_{max}-\delta\epsilon \leq v_{now} \leq t_{max}+\delta\epsilon, ch_i(e!)} S_{next} \ \wedge \ t_{max} - \delta\epsilon \leq (now!v_{now}) \leq t_{max} + \delta\epsilon}{tick!S_{send} \rightarrow ch_i(e!)S_{next}} \tag{D.52}$$

start_component at :

$$\frac{S_{start\_comp} \xrightarrow{tick?, now!, v_{now} > t_{max}+\delta\epsilon} S_{error} \ \wedge \ (now!v_{now}) > t_{max} + \delta\epsilon}{tick!S_{send} \rightarrow S_{error}} \tag{D.53}$$

stop_component at :

$$\frac{S_{stop\_comp} \xrightarrow{tick?, now!, t_{max}-\delta\epsilon \leq v_{now} \leq t_{max}+\delta\epsilon, ch_i(e!)} S_{next} \ \wedge \ t_{max} - \delta\epsilon \leq (now!v_{now}) \leq t_{max} + \delta\epsilon}{tick!S_{send} \rightarrow ch_i(e!)S_{next}} \tag{D.54}$$

stop_component at :

$$\frac{S_{stop\_comp} \xrightarrow{tick?, now!, v_{now} > t_{max}+\delta\epsilon} S_{error} \ \wedge \ (now!v_{now}) > t_{max} + \delta\epsilon}{tick!S_{send} \rightarrow S_{error}} \tag{D.55}$$

Appendix E

Code Examples Using Real-time Extensions Of TTCN-3

This Appendix presents the real-time extensions for TTCN-3, introduced in Chapter 4, by means of examples.

E.1 Data Types Suitable For Expressing Time Values

Listing E.1 presents some usage examples. If the date is incomplete, as in E.1 (Lines 5, 8, 11 - 20), then, the missing value corresponding to year, month, day, etc. will be automatically replaced with the value of year, month day of the current date. The `datetime` type is mainly proposed for distributed test systems, where the synchronization between different component might play an important role. Test components could use `datetime` values as universal synchronization points. `datetime` is also useful for the situation when the tests are run for a long period of time (e.g. more than one day) and the programming of future events depends on previous time values. The domain of `datetime` values will be mathematically represented by the set $\mathbb{DT}$.

Listing E.3 gives examples of `timespan` values. If one `timespan` value contains two terms multiplying the same time unit constant, the expression can be normalized by unifying the respective terms into one, through addition of the corresponding `float` multipliers.

```
var datetime dt1:=1982-02-22T22:10:50_0:0:0 ;                                1
                                                                             2
// this year                                                                 3
var datetime dt2:=02-22T22:10:50_0:0:0=2010-02-220T22:10:50_0:0:0 ;          4
                                                                             5
//this month, this year                                                      6
var datetime dt3:=22T22:10:50_0:0:0=2010-05-22T22:10:50_0:0:0 ;              7
                                                                             8
//this day, this month, this year                                            9
var datetime dt4:=T22:10:50_0:0:0=2010-05-6T22:10:50_0:0:0 ;                 10
                                                                             11
//this hour                                                                  12
var datetime dt5:=T10:50_0:0:0 = 2010-05-6T17:10:50_0:0:0 ;                  13
                                                                             14
//this hour, this minute                                                     15
var datetime dt6:=T50_0:0:0 = 2010-05-6T17:31:50_0:0:0 ;                     16
                                                                             17
//this hour, this minute, this second                                        18
var datetime dt7:=T_0:0:0 = 2010-05-6T17:31:53_0:0:0 ;                       19
```

LISTING E.1: Datetime type values, examples

```
1  var timespan ts_1 := 20.0*min + 10.0*sec ;
2
3  var timespan ts_2 := 20.0*nanosec ;
4
5  var timespan ts_3 := 1.0*hour + 2.0*nanosec ;
6
7  var timespan ts_4 := 10.0*nanosec + 10.0*nanosec;
8
9  /* The values of ts_2, ts_4 and ts_5 are all equal, but
10  * only ts_2 is in the normalized format.
11  */
12 var timespn ts_5 := 15.0*nanosec + 5.0*nanosec;
```

Listing E.2: Timespan type values, examples

```
1  var timespan ts_val_1, ts_val_2, ts_val_3;
2
3  var tick tck_val_1, tck_val_2;
4
5  tck_val_1 := 10;
6
7  ts_val_1 := 2.0 milisec + 1.0 nanosec;
8
9  ts_val_2 := seconds2timespan(ticks2seconds(tck_val_1));
10
11 ts_val_3 := ts_val_1 + ts_val_2 + ts_val_3;
12
13 tck_val_2 := seconds2ticks(timespan2seconds(ts_val_1));
```

Listing E.3: Operations with the new data types example

We can observe in Listing E.3, Lines 3, 7, 12 that the values `ts_2, ts_4` and `ts_5` are all equal, but only `ts_2` is in the normalized format.

There can be defined conversion functions, such as the ones for seconds, at any level of granularity if required. Example of usage for these functions are presented in Listing E.3.

E.2 Special Operations Relaying On Time: `now`, `wait`, `testcasestart`, `testcomponentstart`, `testcomponentstop`

The excerpt of code from Listing E.4 shows some usage scenarios for the presented instructions. The code constitutes a part of a component's behavior and the component may or may not be a main test component (mtc) (Lines 18-19). The time values returned by the introduced instructions are floats and are saved into float variables. The time when the current test case started is saved in variable tf_1 (Line 12), the time when the current test component started is saved in tf_2 (Line 14) and the time when the mtc started is saved in tf_3 (Line 16).

```
1  float tf_1, tf_2, tf_3;
2
3  timespan ts;
4
5  /*
6   * times when the test case, main test component,
7   * the current test component started
8   * are given as floats, indicating the seconds
9   */
10
11 tf_1 = testcasestart;
12
13 tf_2 = self.testcomponentstart;
14
15 tf_3 = mtc.testcomponentstart;
16
17 if(tf_2 == tf_3)
18     log("mtc is self");
19
20 tf_4 = now;
21
22 /*
23  * suspend the execution of the current
24  * test component for 2 seconds
25  */
26
27 wait(2.0);
28
29 tf_5 = now;
30
31 ts=seconds2timespan(tf_5 -    tf_4);
32
33 log("Elapsed time is  ", ts);
```

Listing E.4: Now, wait, testcasestart, testcomponent usage examples

After a waiting time of 2 seconds (Line 28), the execution of the component resumes and the current time is read through the `now` instruction (Line 30). For a proper displaying of the time, it is converted to `timespan` and logged (Lines 32-34).

E.3 `receive` With `timestamp`

In the following, Listing(E.5) exemplifies the usages of this mechanism. The code is showing an alternative statement with three branches. The time stamping instruction is used in addition to the receive statement for the first two branches (Lines 13,22). In the fist case, Line 13, the current time value is saved as `float` and in the second situation, Line 22, the current time value is saved as a `timespan` value.

E.4 `send` With `timestamp`

Usage examples of the `send` with `timestamp` statement are presented in Listing E.6, Lines 12,20. In the first case, the current time value is saved as `float` and in the second case it is saved as `timespan`.

```
float tf_receive;

timespan ts_receive;

alt{
    /*
     * The time value of the message receival is saved as a float
     * indicating the number of seconds passed from the beginning
     * of the testing process, until the moment when the message
     * is entering the system.
     */
    []p_in.receive(MSG_IN_FIRST)->timestamp tf_receive
        {
            log("The message MSG_IN_FIRST was received at second ",
                 tf_receive);
        };
    /*
     * The time value of message receival is saved as a timespan
     * datatype.
     */
    []p_in.receive(MSG_IN_SECOND)->timestamp ts_receive
        {
            log("The message MSG_IN_SECOND was received at ",
                ts_receive);
        };

    []p_in.receive(*){};
}
```

Listing E.5: Receive with timestamp usage example

```
float tf_send;

timespan ts_send;

/*
* The time value when the message is sent is saved as a float
* indicating the number of seconds passed from the beginning
* of the testing process, until the moment when the message
* is leaving the system.
*/
p_out.send(MSG_OUT_FIRST)->timestamp tf_send;

log("The message MSG_OUT_FIRST was send at second ", tf_send);

/*
* The time value when the message is sent is saved as a
* timespan datatype.
*/
p_out.receive(MSG_OUT_SECOND)->timestamp ts_send;

log("The message MSG_OUT_SECOND was send at ", ts_send);
```

Listing E.6: Send with timestamp usage example

```
 1  /* The message is valid when its value conforms to
 2   * MSG_IN_1 and is received in 300 milliseconds
 3   * after the start of the test case
 4   */
 5  p_in.receive(MSG_IN_1)
 6      within (testcasestart..testcasestart+300*millisec);
 7
 8  /* The message is valid when its value conforms to
 9   * MSG_IN_2 and is received anytime before 300 milliseconds
10   * after the start of the test case
11   */
12  p_in.receive(MSG_IN_2)
13      before (testcasestart + 300*millisec);
14
15  /* The message is valid when its value conforms to
16   * MSG_IN_3 and is received anytime after 300 milliseconds
17   * after the start of the test case
18   */
19  p_in.receive(MSG_IN_3)
20      after (testcasestart + 300*millisec);
21
22  /* The message is valid when its value conforms to
23   * MSG_IN_4 and is received approximative 300 milliseconds
24   * after the start of the test case
25   */
26  p_in.receive(MSG_IN_4)
27      at (testcasestart + 300*millisec);
```

Listing E.7: Receive with temporal predicate usage example

```
/* If the time for sending the message is not reached yet,
 * this instruction will be similar with a wait until the
 * moment is reached; afterwards the sending is triggered
 * immediately.
 */

p_out.send(MSG_OUT) at (testcasestart + 300*milisec);
```

Listing E.8: Sending a message at a precise moment, usage example

E.5 `receive` Instructions Which Verify Incoming Communication

The usage of the `receive with time constraints` operation is exemplified in Listing E.7. There four time constraints attached to the four `receive` operations - one for each - which use the four types of basic time predicates (Lines 7, 13-14, 21-22, 27-28). The time parameters are calculated on the basis of the value returned by the `testcasestart` operation.

E.6 `send` Instructions Which Control Outgoing Communication

Listing E.8 presents an example of usage for the `send with time constraints` operation. The time point given as a parameter to the `at` statement should be calculated

```
 1  /* Message MSG_IN is expected in a specified time interval;
 2   * after the expectation time expires without a result,
 3   * the test is ended with a failure;
 4   */
 5
 6  alt{
 7
 8      []p_in.receive(MSG_IN) within
 9          (testcasestart..testcasestart+300*milisec){};
10  }
11  break at(testcasestart+300*milisec){
12
13      setverdict(fail);
14
15      log("The message was not received in time");
16  }
```

Listing E.9: Alt statement with break condition usage example

relative to the moment when the testcase has began. If the value of the provided parameter represents a moment in time that has already passed at the moment when the instruction is executed, then an error routine will be triggered.

E.7 `alt` Instructions which Control Incoming Communication

A usage example of the `alt..break` instruction is presented in Listing E.9. In this case, the alternative has a single branch, meaning that $a_n = 1$. If a valid message is not received within 300 milliseconds from the beginning of the test case execution, then the alternative behavior represented by the block of statements, which follows `break` (Lines 12-17), is going to be executed.

E.8 Instructions Controlling The Starting And Stopping Of Test Components

Listing E.10 provides examples of the usage of instructions which control the start and stop of test components. It can be observed (on Line 6) that the test component has been scheduled to be started 10 milliseconds away from the current time of execution. The current time of execution designates the execution time of the start instruction. The same component is scheduled to be stopped (on Line 10), 100 milliseconds away from the time when the stop instruction has been reached.

```
/* A new test component is created */
var MyComponentType MyComp := MyComponentType.create;
var float ptc_stop, self_stop, all_stop;

/* The new component is started */
MyComp.start(CompBehaviour()) at (now + 10*milisec);
:

/* The component "MyComp" is stopped */
MyComp.stop at (now + 100*milisec);
```

Listing E.10: Start and stop test component at specific time, usage example

Bibliography

[1] Kanaka Juvva. Real-time systems. Technical report, Carnegie Mellon University, 18-849b Dependable Embedded Systems, 1998. URL http://www.ece.cmu.edu/~koopman/des_s99/real_time/#Lui90. Last visited on July 2012.

[2] ETSI: ES 201 873-1 V3.2.1. *Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language*. European Telecommunications Standards Institute, Sophia Antipolis, France, Febr. 2007.

[3] George Din Ina Schieferdecker Diana Alina Serbanescu, Victoria Molovata and Ilja Radusch. Real-time testing with ttcn-3. In *TestCom/FATES*, pages 283–301, 2008.

[4] Juergen Grossmann, Diana Serbanescu, and Ina Schieferdecker. Testing embedded real time systems with ttcn-3. In *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 81–90, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3601-9.

[5] ETSI: ES 201 873-4 V3.2.1. *Methods for Testing and Specification (MTS). The Testing and Test Control Notation version 3, Part 4: TTCN-3 Operational Semantics*. European Telecommunications Standards Institute, Sophia Antipolis, France, Febr. 2007.

[6] Diana Alina Serbanescu and Ina Schieferdecker. Testing environment for real-time communications intensive systems. In *Proceedings of the 2010 Sixth International Conference on Networking and Services*, ICNS '10, pages 368–374, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-3969-0. URL http://dx.doi.org/10.1109/ICNS.2010.58. Last visited on July 2012.

[7] Frederick M. Proctor and William P. Shackleford. Real-time operating system timing jitter and its impact on motor control, 2001.

[8] Patrick Donohoe. A survey of real-time performance benchmarks for the ada programming language. Technical report, Carnegie Mellon University, 1987. URL www.sei.cmu.edu/reports/87tr028.pdf. Last visited on April 2013.

[9] William R. Bush. Effects of compiler and runtime system features on embedded system designs, 1990.

[10] ITU-T. Itu telecommunication standardization sector. URL http://www.itu.int/en/ITU-T/Pages/default.aspx. Last verified on April 2013.

[11] Laurie Williams. Testing overview and black-box testing techniques. Technical report, 2006. URL http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf. Last visited on April 2013.

[12] Paul Pettersson. Modelling and verification of real-time systems using timed automata: Theory and practice. Technical report, Uppsala University, 1999. URL http://user.it.uu.se/~paupet/thesis.shtml. Last visited on April 2013.

[13] Wind River. Wind river's vxworks powers mars science laboratory rover, curiosity, 2013. URL http://www.windriver.com/news/press/pr.html?ID=10901. Last visited on April 2013.

[14] Kevin M. Obenland. The use of posix in real-time systems, assessing its effectiveness and performance, 2000. URL www.mitre.org/work/tech_papers/tech...00/.../obenland_posix.pdf. Last visited on April 2013.

[15] Nels Beckman. A little bit of real-time java, 2005. URL www.cs.cmu.edu/~nbeckman/presentations/a_little_bit_of_rt_java.ppt. Last visited on April 2013.

[16] Kelvin Nilsen. Exploiting real-time java hierarchies to improve generality and performance, 2007. URL http://www.sstc-online.org/2007/pdfs/KN1660.pdf. Last visited on April 2013.

[17] AlexanderD. Stoyenko. Real-time euclid: Concepts useful for the further development of pearl. In Wilfried Gerth and Per Baacke, editors, *PEARL 90 Workshop ber Realzeitsysteme*, volume 262 of *Informatik-Fachberichte*, pages 1–11. Springer Berlin Heidelberg, 1990. ISBN 978-3-540-53464-8. doi: 10.1007/978-3-642-46725-7_1. URL http://dx.doi.org/10.1007/978-3-642-46725-7_1.

[18] Richard Goering. Ada 2005 speaks to real-time embedded applications. Technical report, EETimes, 2007. URL http://eetimes.com/electronics-news/4070784/Ada-2005-speaks-to-real-time-embedded-applications. Last visited on April 2013.

[19] M. Schiebe and S. Pferrer. *Real-Time Systems Engineering and Applications*. The Springer International Series in Engineering and Computer Science. Springer, 1992. ISBN 9780792391968. URL http://books.google.de/books?id=x6BQAAAAMAAJ.

[20] Wikibooks contributors. Ada programmins. Technical report, Wikibooks, 2004-2007. URL http://en.wikibooks.org/wiki/Ada_Programming. Last visited on April 2013.

[21] TimeSys Corporation. The concise handbook of real-time systems. Technical Report Pittsburgh, PA, TimeSys Corporation, 2002.

[22] C. J. Fidge. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems*, 14:61–93, 1998.

[23] Verifysoft Technology. Advantages of ttcn-3. URL http://www.verifysoft.com/en_TTCN-3.html. Last visited on March 2013.

[24] Testing Technologies. Benefits of ttcn-3, . URL http://www.testingtech.com/ttcn3/benefits.php. Last visited on March 2013.

[25] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, December 1990.

[26] Raimund Kirner and Peter Puschner. Time-predictable computing. In *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'10, pages 23–34, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16255-X, 978-3-642-16255-8. URL http://dl.acm.org/citation.cfm?id=1927882.1927890.

[27] George Anwar. Real-time application of mini and micro computer. Technical report, UC Berkeley, Fall 2008.

[28] John A. Stankovic. Real-time and embedded systems. In *The Computer Science and Engineering Handbook*, pages 1709–1724. 1997.

[29] S. Voget. Future trends in software architectures for automotive systems. Technical Report Berlin, Germany, Microsystems for Automotive Applications, 2003.

[30] CAR 2 CAR Communication Consortium. Concerned with the development and release of an open european standard for cooperative intelligent transport systems and associated validation process with focus on inter-vehicle communication systems. URL http://www.car-to-car.org/. Last visited on July 2012.

[31] Manfred Broy. Challenges in automotive software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. Last visited on July 2012.

[32] Mirko Conrad. M.: Systematic testing of embedded automotive software - the classification-tree method for embedded systems. proc. dagstuhl seminar n 04371 'perspectives of model-based testing', schlo dagstuhl (d). In *Software Tests, SAE World Congress 2005, Detroit (US)*. Wiley, 2005.

[33] Computing Students. Computing students is in constant development with new content being added regularly, provides definitions of relevant terms in computer science. URL http://www.computingstudents.com/notes/software_analysis/classification_tree_method.php. Last visited on July 2012.

[34] Wikipedia. Tessy tool for embedded systems testing., . URL http://en.wikipedia.org/wiki/Tessy_(Software). Last visited on July 2012.

[35] mTest classic. Efficient model test management for simulink and targetlink models. URL http://www.mtest-classic.com/. Last visited on July 2012.

[36] MathWorks. Simulink is an environment for multidomain simulation and model-based design for dynamic and embedded systems. URL http://www.mathworks.co.uk/products/simulink/. Last visited on July 2012.

[37] Klaus Grimm. Software technology in an automotive company: major challenges. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 498–503, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X.

[38] Johannes Slettengren. Automated testing in model based design-an implementation for cruise control manager. Technical report, Scania CV AB, Systems and Software Departament, Sweden.

[39] dSPACE. dspace training and engineering. URL http://dspaceinc.com/en/inc/home/products/sw/expsoft/automdesk/testautomation.cfm. Last visited on July 2012.

[40] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21:10–19, October 1988. ISSN 0018-9162. URL http://portal.acm.org/citation.cfm?id=50810.50811. Last visited on July 2012.

[41] Phillip A. Laplante. *Real-time systems design and analysis - an engineer's handbook (2. ed.)*. IEEE, 1997. ISBN 978-0-7803-3400-7.

[42] Srini Vasan Bruce Powel Douglass. *Timeliness and Performance in Real-Time Object Designs*. TimeSys Corp. URL http://www.swd.ru/files/share/Rhapsody/materials/Whitepapers/Timeliness.pdf. Last visited on July 2012.

[43] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison Wesley, fourth edition, May 2009. ISBN 0321417453. URL http://www.amazon.com/exec/obidos/

redirect?tag=citeulike07-20&path=ASIN/0321417453. Last visited on July 2012.

[44] John Barnes. *Introducing Ada95*. 1995. URL http://www.adapower.com/launch.php?URL=http%3A%2F%2Fwww.seas.gwu.edu%2F~adagroup%2Fsigada-website%2Fbarnes-html%2Fintro.html. Last visited on July 2012.

[45] Eugene Kligerman and Alexander D. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Trans. Software Eng.*, pages 941–949, 1986.

[46] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201703238.

[47] Kevin M. Obenland. Posix in real-time. Technical report, 3/15/2001. URL http://www.eetimes.com/electronics-news/4133374/POSIX-in-Real-Time. Last visited on July 2012.

[48] Dennis Ludwig. An introduction to real-time programming. Technical report, Aeronautical Systems Center. URL http://www.docstoc.com/docs/29480263/An-Introduction-to-Real-Time-Programming#. Last visited on July 2012.

[49] Wikipedia. Worst-case execution time definition. Technical report, . URL http://en.wikipedia.org/wiki/Worst-case_execution_time. Last visited on July 2012.

[50] T. Straumann. *Open Source Real-Time Operating Systems Overview*. 2001. URL http://www.slac.stanford.edu/econf/C011127/WEBI001.pdf. Last visited on July 2012.

[51] D. Kalinsky. *Basic Concepts of Real-Time Operating Systems*. 2003. URL http://linuxdevices.com/articles/AT4627965573.html. Last visited on May 2011.

[52] Jia Xu and David Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Trans. Softw. Eng.*, 16:360–369, March 1990. ISSN 0098-5589. URL http://portal.acm.org/citation.cfm?id=78266.78285. Last visited on July 2012.

[53] R. Lui Sha, Rajkumar and S.S. Sathaye. Generalized rate-monotonic scheduling theory: a framework for developing real-time systems. *Proceedings of the IEEE*, 82:68–82, August 2002. ISSN 0018-9219.

[54] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[55] Yde Venema, Lou Goble (ed, Blackwell Guide, Philosophical Logic, and Blackwell Publishers. Temporal logic. In *The Blackwell Guide to Philosophical Logic. Blackwell Philosophy Guides (2001.* Basil Blackwell Publishers, 1998.

[56] Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-25813-2.

[57] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999. ISBN 0471623733.

[58] J. Davies. *Specification and Proof in Real-time CSP.* Cambridge University Press, 1993.

[59] Gerd Behrmann, Re David, and Kim G. Larsen. *A tutorial on uppaal.* Springer, 2004.

[60] ETSI: ES 201 873-2 V3.2.1. *Methods for Testing and Specification (MTS). The Testing and Test Control Notation version 3, Part 2: TTCN-3 Tabular presentation Format (TFT).* European Telecommunications Standards Institute, Sophia Antipolis, France, Febr. 2007.

[61] ETSI: ES 201 873-3 V3.2.1. *Methods for Testing and Specification (MTS). The Testing and Test Control Notation version 3, Part 3: TTCN-3 Graphical presentation Format (GFT).* European Telecommunications Standards Institute, Sophia Antipolis, France, Febr. 2007.

[62] ETSI: ES 201 873-5 V3.2.1. *Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 5: TTCN-3 Runtime Interfaces.* European Telecommunications Standards Institute, Sophia Antipolis, France, Febr. 2007.

[63] ETSI: ES 201 873-6 V3.2.1. *Methods for Testing and Specification (MTS). The Testing and Test Control Notation version 3, Part 6: TTCN-3 Control Interface (TCI).* European Telecommunications Standards Institute, Sophia Antipolis, France, Febr. 2007.

[64] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (ttcn-3). *Computer Networks*, 42(3):375–403, 2003.

[65] Stephan Tobies Stefan Keil Federico Engler Stephan Schulz Anthony Wiles Colin Willcock, Thomas Dei. WILEY, February 2011. ISBN 978-0-470-66306-6.

[66] Joachim Wegener, Klaus Grimm, Matthias Grochtmann, and Harmen Sthamer. Eurostar96 systematic testing of real-time systems. 1996.

[67] Jos C. M. Baeten and Sjouke Mauw, editors. *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, 1999. Springer. ISBN 3-540-66425-4.

[68] Marius Bozga, Jean-Claude Fernandez, Lucian Chirvu, Susanne Graf, Jean pierre Krimm, and Laurent Mounier. If: A validation environment for timed asynchronous systems, 2000.

[69] Jean-Claude Fernandez, Claude Jard, Thierry Jron, and Csar Viho. Using on-the-fly verification techniques for the generation of test suites, 1996.

[70] Holger Krisp, Klaus Lamberg, and Robert Leinfellner. Automated real-time testing of electronic control units, 2007.

[71] BasilDev. Testfarm core automated testing system platform. URL http://www.testfarm.org/documents/IP060014-en%20TestFarm%20Core%20Product%20Brief.2.pdf. Last verified on July 2012.

[72] ReACT Technologies. Talent, technical features, . URL http://www.reacttech.com/Talent.htm. Last verified on July 2012.

[73] Applied Dynamics International. Test automation with the advantage simulation framework. URL http://www.adi.com/products_sis.htm. Last verified on July 2012.

[74] Ina Schieferdecker, Bernard Stepien, and Axel Rennoch. Perfttcn, a ttcn language extension for performance testing. URL http://www.site.uottawa.ca/~bernard/PerfTTCN.pdf. Last verified on July 2012.

[75] Thomas Walter and Jens Grabowski. *Real-time TTCN for testing real-time and multimedia systems.* September 1997.

[76] Z. Dai, J. Grabowski, and H. Neukirchen. *Timed TTCN-3 – A Real-Time Extension for TTCN.* 2002. URL citeseer.ist.psu.edu/article/dai02timedttcn.html. Last verified on July 2012.

[77] Ina Schieferdecker and Juergen Grossmann. Testing embedded control systems with ttcn-3: an overview on ttcn-3 continuous. pages 125–136, 2007. URL http://portal.acm.org/citation.cfm?id=1778978.1778994. Last verified on July 2012.

[78] R.O. Sinnott. Towards more accurate real-time testing. In *The 12th International Conference on Information Systems Analysis and Synthesis (ISAS 2006)*, Orlando, Florida, 2006.

[79] TEMEA Project. web site of temea (test specification and test methodology for embedded systems in automobiles), 2008. URL http://www.temea.org. Last verified on July 2012.

[80] *Methods for Testing and Specification (MTS). The Testing and Test Control Notation version 3 .TTCN-3 Language Extensions: TTCN-3 Performance and Real Time Testing*. European Telecommunications Standards Institute, 2010.

[81] Helmut Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, Georg-August-Universitt Gttingen, 2004.

[82] Mirko Conrad. *Modell-basierter Test eingebetteter Software im Automobil*. PhD thesis, TU-Berlin, 2004.

[83] Ina Schieferdecker Jürgen Großmann and Hans-Werner Wiesbrock. Modeling property-based stream templates with ttcn-3. In Kenji Suzuki and Hasegawa [84], pages 70–85. ISBN 978-3-540-68514-2.

[84] Andreas Ulrich Kenji Suzuki, Teruo Higashino and Toru Hasegawa, editors. *Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop, FATES 2008, Tokyo, Japan, June 10-13, 2008, Proceedings*, volume 5047 of *Lecture Notes in Computer Science*, 2008. Springer. ISBN 978-3-540-68514-2.

[85] The MathWorks. The mathworks helpdesk - Stateflow Notation, 2007. URL http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/ug/f18-32361.html. Last verified on May 2011.

[86] AUTOSAR. Autosar (automotive open system architecture) is a worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry., 2008. URL http://www.autosar.org. Last verified on July 2012.

[87] Ina Schieferdecker and Jrgen Gromann. Testing of Embedded Control Systems with Continous Signals. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme II*, pages 113–122. TU Braunschweig, 2006.

[88] Ina Schieferdecker, Eckard Bringmann, and Juergen Grossmann. Continuous ttcn-3: Testing of embedded control systems. In *SEAS '06: Proceedings of the 2006*

international workshop on Software engineering for automotive systems, pages 29–36, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-402-2.

[89] Leon Osterweil. Strategic directions in software quality. *ACM Comput. Surv.*, 28 (4):738–750, 1996. ISSN 0360-0300.

[90] T. Zurawka J. Schuffele, editor. *Automotive Software Engineering.* Vieweg & Sohn Verlag, Wiesbaden, 2006. ISBN 978-3528010409.

[91] Holger Giese and Sven Burmester. Real-time statechart semantics. Technical Report tr-ri-03-239, Lehrstuhl fr Softwaretechnik, Universitt Paderborn, Paderborn, Germany, 6 2003.

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

[93] Marius Mikucionis Brian Nielsen Paul Pettersson Anders Hessel, Kim Guldstrand Larsen and Arne Skou. Testing real-time systems using uppaal. In *Formal Methods and Testing*, pages 77–117, 2008.

[94] E. Mikucionis and E. Sasnauskaite. On-the-fly testing using uppaal. Master's thesis, Departament of Computer Science Aalborg Univeristy, Denmark, June 2003.

[95] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. Online on-the-fly testing of real-time systems. Technical report, December 2003.

[96] B. Nielsen E. Brinksma, K. Larsen and J. Tretmans. Systematic testing of realtime embedded software systems (stress). Research proposal submitted and accepted by the Dutch Research Council, March 2002.

[97] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design ofComputer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 281–297, London, UK, 1998. Springer-Verlag. ISBN 3-540-64356-7.

[99] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*,

pages 299–306, New York, NY, USA, 2005. ACM. ISBN 1-59593-091-4. URL http://doi.acm.org/10.1145/1086228.1086283. Last visited on July 2012.

[100] Anders Hesselz Kim, Kim G. Larseny, Brian Nielseny, Paul Petterssonz, and Arne Skouy. Time-optimal real-time test case generation using uppaal. In *in Proc. of the 3rd Intl. Workshop on Formal Approaches to Testing of Software (FATES'03)*, 2003.

[101] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[102] Ichiro Satoh and Mario Tokoro. Semantics for a real-time object-oriented programming language. In *ICCL*, pages 159–170, 1994.

[103] Jos C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335 (2-3):131–146, 2005.

[104] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *REX Workshop*, pages 226–251, 1991.

[105] Holger Giese and Sven Burmester. Real-time statechart semantics. In *Self-optimizing Concepts and Structures in mechanical Engineering*, 2003.

[106] Hardi Hungar Werner Damm, Bernhard Josko and Amir Pnueli. A compositional real-time semantics of statemate designs. In *COMPOS*, pages 186–238, 1997.

[107] Peter P. Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

[108] Peter P. Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[109] Bernhard Rieder Ingomar Wenzel, Raimund Kirner and Peter P. Puschner. Cross-platform verification framework for embedded systems. In *SEUS*, pages 137–148, 2007.

[110] Moez Krichen and Stavros Tripakis. Real-time testing with timed automata testers and coverage criteria. Technical Report TR-2004-15, Verimag Technical Report, 2004.

[111] Wolfgang A. Halang and Alexander D. Stoyenko. *Constructing Predictable Real Time Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1991. ISBN 0792392027.

[112] ISO. *ISO 8601:2004. Data elements and interchange formats – Information interchange – Representation of dates and times.* 3 edition, 2004. URL http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=01&ics2=140&ics3=30&csnumber=40874. Last visited on July 2012.

[113] Liu Yang. Pat: Process analysis toolkit. an enhanced simulator, model checker and refinement checker for concurrent and real-time systems., 2007. URL http://www.comp.nus.edu.sg/~pat/. Last visited on July 2012.

[114] Department of Information Technology at Uppsala University (UPP) in Sweden and the Department of Computer Science at Aalborg University (AAL) in Denmark. Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). URL http://www.uppaal.org/. Last visited on July 2012.

[115] FreeRTOS Mantainance Team. Web pages of FreeRTOS - The FreeRTOS Project, . URL http://www.freertos.org/. Last visited on July 2012.

[116] RTAI Project Mantainance Team. Web pages of RTAI - Real Time Application Interface for Linux, . URL https://www.rtai.org/. Last visited on July 2012.

[117] eCos. ecos is a free open source real-time operating system intended for embedded applications. URL www.ecos.sourceware.org. Last visited on July 2012.

[118] Ralf Corsepius Chris Johns Eric Norum Joel Sherrill, Jennifer Averett and Thomas Doerfler. Rtems operating systems. the real-time executive for multiprocessor systems or rtems is a full featured rtos that supports a variety of open api and interface standards. URL www.rtems.com. Last visited on July 2012.

[119] Hard Real-Time Networking for Real-Time Linux. Rtnet is an open soure hard real-time network protocol stack for xenomai and rtai (real-time linux extensions). URL http://www.rtnet.org/. Last visited on July 2012.

[120] freescale. Mc9s12ne64 demonstration kit. URL http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=DEMO9S12NE64. Last visited on July 2012.

[121] Open Source Community. Open source watcom c, c++, and fortran cross compilers and tools. URL www.openwatcom.org. Last visited on July 2012.

[122] Dedicated Systems Encyclopedia. The embedded systems' products, services & research guide. URL www.dedicated-systems.com/encyc/buyersguide/products/Dir1048.html. Last visited on July 2012.

[123] Internet FAQ Archives. Comp.realtime: A list of commercial real-time operating systems. *www.faqs.org/faqs/realtime-computing/list/*. Last visited on July 2012.

[124] Juergen Grossmann, Diana Serbanescu, and Ina Schieferdecker. Testing embedded real time systems with ttcn-3. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:81–90, 2009.

[125] Glenford J. Myers. *The Art of Testing*. John Wiley & Sons, first edition, 1979. ISBN 0-471-04328-1.

[126] Moez Krichen and Stavros Tripakis. Real-time testing with timed automata testers and coverage criteria. Technical Report TR-2004-15, Verimag Technical Report, 2004.

[127] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. T-UPPAAL: online model-based testing of real-time systems. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 396–397, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2131-2.

[128] Rachel Cardell-oliver and Tim Glover. A practical and complete algorithm for testing real-time systems. In *In Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 1486*, pages 251–261. Springer-Verlag, 1998.

[129] Duncan Clarke and Insup Lee. Testing real-time constraints in a process algebraic setting. In *In International Conference on Software Engineering*, pages 51–60, 1995.

[130] Simon Hill and Bala Krishnamurthy. Introduction to linux for real-time control. Technical report, prepared for NIST by Aeolean Inc., 2002. Introductory Guidelines and Reference for Control Engineers and Managers.

[131] Joseph Leung and Hairong Zhao. Real-time scheduling analysis. Technical Report DOT/FAA/AR-05/27, Departament of Computer Science New Jersey Institute of Technology, 2005.

[132] Paul Seidel. Realtime linux. URL `http://www.dcl.hpi.uni-potsdam.de/teaching/proccontrol/slides/RTLinux.pdf`. Last visited on May 2011.

[133] David Beal. Real-time application interface (rtai) for linux. URL `http://opengroup.org/rtforum/feb2001/slides/beal.pdf`. Last visited on July 2012.

[134] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell, 1997.

[135] Luis E. Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. In *RTCSA '06: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 385–394, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2676-4.

[136] Frederick M. Proctor and William P. Shackleford. Real-time operating system timing jitter and its impact on motor control. In *RTCSA '06: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. National Institute of standards and Technology, 2001.

[137] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of the IEEE*, pages 55–67, 1994.

[138] Arezou Mohammadi and Selim G. Akl. Scheduling algorithms for real-time systems. Technical report, School of Computing Queen's University, 2005.

[139] Peter Jay Salzman. *The Linux Kernel Module Programming Guide*. CreateSpace, Paramount, CA, 2009. ISBN 1441418865, 9781441418869.