Modeling, Analysis and Verification of Real-Time Software Systems: An Integrated Approach

Thesis submitted by

Rumpa Hazra

Doctor of Philosophy (Engineering)

Department of Computer Science and Engineering

Faculty Council of Engineering and Technology

Jadavpur University

Kolkata, India

2019

JADAVPUR UNIVERSITY
KOLKATA-700032, INDIA

INDEX NO.:265/14/E

1. **Title of the Thesis:** Modeling, Analysis and Verification of Real-Time Software Systems: An Integrated Approach

2. **Supervisors:**

- **Dr. Swapan Bhattacharya**
  Professor,
  Department of Computer Science & Engineering,
  Jadavpur University, Kolkata-700032, India
  Email: bswapan2000@yahoo.co.in

- **Dr. Ananya Kanjilal**
  Associate Professor,
  Department of Computer Science & Engineering,
  B.P. Poddar Institute of Management and Technology, Kolkata-700052, India
  Email: ag_k@rediffmail.com

3. **List of Publication:**

   (a) Rumpa Hazra, Shouvik Dey, Ananya Kanjilal, Swapan Bhattacharya, "Comparative Analysis of Real Time Resource Access Control Protocols using UML 2.0", ACM SIGSOFT Software Engineering Notes (SEN), vol. 38, issue 5, pp. 1-7, September, 2013.

   (b) Rumpa Hazra, Shouvik Dey, Ananya Kanjilal, Swapan Bhattacharya, "Modeling and Analysis of Real Time Fixed Priority Scheduling using UML 2.0", INFOCOMP Journal of Computer Science, vol. 12, issue 1, pp. 36-48, June 2013.

   (c) Shouvik Dey, Rumpa Hazra, Ananya Kanjilal, Swapan Bhattacharya, "Automated consistency checking of UML/MARTE based software systems", In Proceedings of TENCON 2018 - 2018 IEEE Region 10 Conference, pp. 2259-2264, Jeju, Korea, 28-31st October 2018.

   (d) Rumpa Hazra, Shouvik Dey, Ananya Kanjilal, Swapan Bhattacharya, "Modeling and Analysis of Deadlock Driven Dynamic Priority Scheduling", In Proceedings of TENCON 2013 - 2013 IEEE Region 10 Conference, pp. 1-4, China, 22-25th October 2013.

4. **List of Patents:** NIL

5. **List of Presentations in National/ International/ Conferences/ Workshops:**

   (a) Shouvik Dey, Rumpa Hazra, Ananya Kanjilal, Swapan Bhattacharya, "Automated consistency checking of UML/MARTE based software systems", In Proceedings of TENCON 2018 - 2018 IEEE Region 10 Conference, pp. 2259-2264, Jeju, Korea, 28-31st October 2018.

   (b) Rumpa Hazra, Shouvik Dey, Ananya Kanjilal, Swapan Bhattacharya, "Modeling and Analysis of Deadlock Driven Dynamic Priority Scheduling", In Proceedings of TENCON 2013 - 2013 IEEE Region 10 Conference, pp. 1-4, China, 22-25th October 2013.

Certificate from the Supervisors

This is to certify that the thesis entitled "Modeling, Analysis and Verification of Real Time Software Systems: An Integrated Approach" submitted by Smt. Rumpa Hazra, who got her name registered on 29th January, 2014 for the award of Ph.D. (Engg.) degree of Jadavpur University is absolutely based upon her own work under the supervision of Prof. Swapan Bhattacharya and Dr. Ananya Kanjilal and that neither her thesis nor any part of the thesis has been submitted for any degree/diploma on any other academic award anywhere before.

[illegible] 21.01.2019

Signature of the Supervisor
(Prof. Swapan Bhattacharya)

Professor
Computer Sc. & Engg. Department
Jadavpur University
Kolkata-700032

[illegible] 21.1.19

Signature of the Supervisor
(Dr. Ananya Kanjilal)

HOD
Dept. of Computer Science And Engineering
B. P. Poddar Institute of Management And Technology
137, V.I.P. Road. Kolkata--700 052

Acknowledgement

I would like to express my sincerest gratitude to my supervisors Professor Swapan Bhattacharya and Dr. Ananya Kanjilal for their valuable suggestions and support through out my research work. This dissertation would have not been made possible without their continuous motivation, encouragement and overall guidance.

I am really grateful to my parents for their constant inspiration who have supported and helped me in every possible way throughout the period of this work.

I cannot thank enough my husband who has stood by me every moment and has helped me to continue with my work, provided me with useful suggestions, motivation and also constant support and encouragement to complete this dissertation.

Contents

List of Figures

List of Tables

Abbreviations

RTS	**R**eal-**T**ime **S**ystem
RTSS	**R**eal-**T**ime **S**oftware **S**ystem
RTES	**R**eal-**T**ime **E**mbedded **S**ystem
OMG	**O**bject **M**anagement **G**roup
OMT	**O**bject **M**odeling **T**echnique
UML	**U**nified **M**odeling **L**anguage
MARTE	**M**odeling and **A**nalysis of **R**eal-**T**ime **E**mbedded Systems
OCL	**O**bject **C**onstraint **L**anguage
MOF	**M**eta **O**bject **F**acility
VSL	**V**alue **S**pecification **L**anguage
NFP	**N**on **F**unctional **P**roperty
PVS	**P**atient **V**entilation **S**ystem
CFG	**C**ontext **F**ree **G**rammar
ANTLR	**AN**other **T**ool for **L**anguage **R**ecognizer
EDF	**E**arliest **D**eadline **F**irst
PIP	**P**riority **I**nheritance **P**rotocol
PCP	**P**riority **C**eiling **P**rotocol
SPCP	**S**tack **b**ased **P**riority **C**eiling **P**rotocol
SBPCP	**S**tack **B**ased **P**reemption **C**eiling **P**rotocol
SRP	**S**hared **R**esource **P**olicy

Symbols

R	Set of requirements
UC	Set of Use case diagrams
U	Set of Use cases in Use case diagram
AU	Set of actors in Use case diagram
SU	Set of stereotypes in Use case diagram
CU	Set of timing constraints in Use case diagram
SQ	Set of Sequence diagrams
AS	Set of actors in Sequence diagram
PS	Set of lifelines in Sequence diagram
MSS	Set of messages in Sequence diagram
MS	Set of methods in Sequence diagram
SS	Set of stereotypes in Sequence diagram
CS	Set of timing constraints in Sequence diagram
TM	Set of timing diagrams
PT	Set of Participants in Timing diagram
MST	Set of messages in Timing diagram
MT	Set of methods in Timing diagram
ST	Set of stereotypes in Timing diagram
CT	Set of timing constraints in Timing diagram
CL	Set of Class diagrams
C	Set of Classes in Class diagram
MC	Set of Methods in Class diagram

SC	Set of stereotypes in Class diagram
CC	Set of timing constraints in Class diagram
N(S)	Number of elements present in the set S

Abstract

Abstract

Real-Time Systems (RTS) interact with their environment using time constrained input/output signals i.e. the systems process information and produce responses within a specified time otherwise failure of the systems may occur. The complexity of Real-Time Software Systems (RTSS) is continuously increasing which makes their design very challenging. Therefore, ensuring the correctness of such systems within the specified time constraints is a difficult and complex task. In this dissertation, we have presented some new ideas and developed some novel approaches for modeling, analysis and verification of RTSS through the different phases of development. We have considered some of the commonly used UML diagrams to develop our methodology. The model-driven approach is adequate to address the complex issues of RTSS. This enables the verification of design and potentially the automatic synthesis of implementations. We have developed a framework, which focuses on the modeling of real-time software development to capture different aspects of RTSS. We have introduced new user defined data types with VSL and then addressed the representation of different UML diagrams by extending the MARTE metamodel.

Software life cycle model consists of different interrelated stages, due to which, there is a requirement to verify inconsistencies among these stages. By checking consistency, we ensure that the properties of the different stages of a specification are consistent with each other and they do not oppose. The same holds for RTSS as well. In order to make our model more robust, we have developed a set of consistency rules within the design models and introduced formal approach for the automated verification of the consistency rules.

A prime concern of RTSS is that all the requirements related to time must be traced in all the phases of software development life cycle consistently. We have developed a comprehensive framework for ensuring traceability of timing constraints from the requirements analysis phase into the design phase. This framework helps in verification of requirements in design, automatically generates the trace metrics based on UML diagrams, demonstrates the degree of coverage of timing requirements and finally features any missing requirements.

In RTS, scheduling of tasks with hard deadlines has been an important area of research. The trouble of giving a guarantee of meeting hard deadlines of tasks lies in the issues of priority inversion and of deadlocks. To beat such difficulties, a great resource access control protocol is required. To address this issue, we model, analyze and verify four existing protocols (Priority Inheritance Protocol, Priority Ceiling Protocol, Stack Based Priority Ceiling Protocol and Stack Based Preemption Ceiling Protocol) using UML/SPT based Sequence and Timing diagrams to study deadlock. We have focused on the occurrence of deadlock in the Priority Inheritance Protocol and prevention of this using other mentioned protocols. The methods and frameworks presented in this dissertation provide an integrated framework towards modeling, analysis and verification of RTSS.

Chapter 1

Introduction

1.1 Real-Time Systems

The behaviors of Real-Time Systems (RTS) do not depend only on the values of input and output signals, but also on their time of occurrence. RTS interact with their environment using time constrained input/output signals i.e. the systems process information and produce responses within a specified time otherwise failure of the systems may occur. RTS are now omnipresent in modern societies in various domains such as patient monitoring systems, air traffic control systems, avionics, control of nuclear power stations, multimedia communications, robotics, process control, embedded and telecommunication systems. For such systems, a functional misbehavior or a deviation from the specified time constraints may have catastrophic consequences.

The functions performed by the real-time systems are uniformly, executed by a fixed number of tasks. These systems can be categorized as hard and soft. A hard RTS requires that a result must be produced within a bounded interval otherwise a serious fault is said to occur. In a soft RTS occasional timing faults may be permitted. Examples of soft RTS are video play back system, on line transaction system, telephone switches as well as electronic games. Hard Real-Time Systems are used in a wide range of mission-critical applications such as avionics systems, aerospace systems, robotics and defense systems.

RTS is often driven by hardware, software architecture, operating system characteristics, application requirements, programming language as well as design issues.

1.2 Real-Time Software Systems

The design of real-time software encompasses all aspects of conventional software design while at the same time introducing a new set of design criteria and concerns. The design of real-time software systems is the most challenging and complex task that can be undertaken by a software engineer. By its very nature, software for real-time systems makes demands on analysis, design and verification techniques. Real-time software programs can be found in various applications. Some of them are anti-virus programs, which perform scheduled maintenance checks, as well as database applications like airline database controls and 24-hour transaction facilities. A real-time operating system (RTOS) is an operating system that is implemented for real-time systems in order to simplify design, execution and maintenance of real-time systems and applications. The RTOS provides the designer with a programming interface to the underlying hardware [38].

Complexity of RTSS is continuously increasing which makes their design very challenging. Therefore, ensuring the correctness of such systems within the specified time constraints is a difficult and complex task. In order to cope with the complexity of RTSS, there is a high requirement to follow a model driven approach such as the Model-Driven Architecture (MDA), which relies on using models of high level abstraction for the development of such systems. In this dissertation, some new ideas in the domain of modeling, consistency, verification and schedulabity analysis for RTSS have been presented.

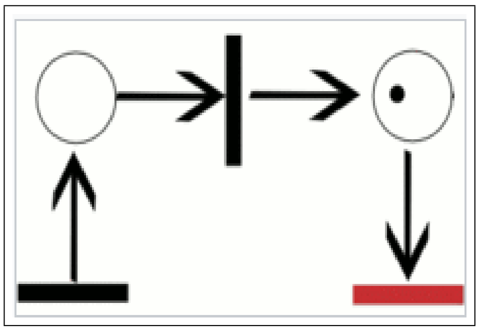


FIGURE 1.1: Petri Net

1.3 Modeling techniques for Real-Time Software Systems

Several modeling techniques are available for RTSS such as Petri net, Timed automata, Unified Modeling Language. A Petri net (shown in Figure 1.1), also known as a place/transition (PT) net, is one of several mathematical modeling languages for the description of RTS. Petri Net was developed originally by Carl Adam Petri [97] and was the subject of his dissertation in 1962.

It is a class of discrete event dynamic system. A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, represented by bars) and places represent conditions (represented by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows). Like industry standards such as UML activity diagrams and Business Process Model, Petri nets offer a graphical notation for stepwise processes that include choice, iteration, and concurrent execution. Unlike these standards, Petri nets have an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis.

Timed automaton [4] is a technique for modeling and verification of RTS, which is essentially a finite automaton (that is a graph containing a finite set of nodes and a finite set of labeled edges) extended with real-valued variables. Such an automaton

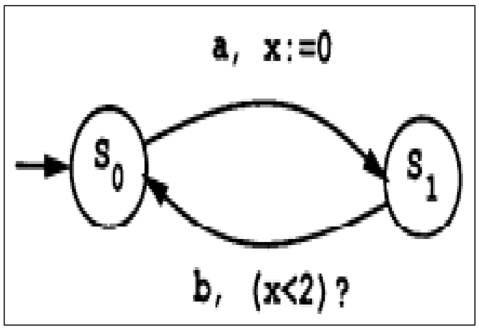


FIGURE 1.2: Timed Automata

may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then increase synchronously with the same rate.

Formally, a timed automaton is a tuple

A = (Q,Σ,C,E,q0) that consists of the following components:

Q is a finite set. The elements of Q are called the states of A.

Σ is a finite set called the alphabet or actions of A.

C is a finite set called the clocks of A.

E ⊆ Q × Σ × B(C) × P(C) × Q is a set of edges, called transitions of A, where

B(C) is the set of boolean clock constraints involving clocks from C, and

P(C) is the powerset of C.

q0 is an element of Q, called the initial state.

An edge (q,a,g,r,q') from E is a transition from state q to q' with action a, guard g and clock resets r.

The automaton in Figure(1.2) starts in state s_0, and moves to state s_1 reading the input symbol a. The clock x gets set to 0 along with this transition. While in state s_1, the value of the clock x shows the time elapsed since the occurrence of the last a symbol. The transition from state s_1 to s_0 is enabled only if this value is less than 2. The whole cycle repeats when the automaton moves back to state s_0.

The Unified Modeling Language (UML) is the Object Management Group (OMG) standard modeling language to support MDA. UML is appropriate for software systems because it allows for a multi view modeling approach through its multitude of diagrams covering the structure, the behavior and the deployment architecture.

UML encourages the use of automated tools that facilitate the development process from analysis through coding. This is particularly true for real-time embedded systems also, whose behavioral aspects can often be described via UML. It is therefore interesting to consider how well UML is adapted to the real-time context. One important feature of UML stems from its built-in extensibility mechanisms: stereotypes, tag values and profiles. These allow adapting UML to fit the specifics of particular domains or to support a specific analysis [77].

As a real-time modeling language, UML additionally has a few difficulties. It is furnished with a variety of syntax but lacks adequate semantics. It is only a visual language with diagrams and not a programming language. As software is quickly developing in degree and intricacy, graphical representation using UML should be formally confirmed, before the execution stage, on request to ensure the improvement of more solid frameworks.

However, UML is not adequate to model domain-specific constructs. In order to fill up this gap, different organizations have extended UML for modeling RTSS. For example, IBM has developed UML-RT. OMG has introduced UML/SPT, UML/QoS and UML/MARTE etc. UML-RT can model the structure and the behavior of real-time systems, but it does not support time and timing constraints modeling. In comparison to this, UML/SPT offers concepts for modeling of time as well as concurrency, but it is not sufficient to capture real-time qualitative features and lacks expressive power and flexibility. In order to overcome those difficulties, UML/MARTE is introduced. MARTE provides predefined stereotypes and tagged values to specify non-functional properties, time related constraints, general resources (software and hardware). Those modeling artifacts are essential concepts to design real-time software systems.

Modeling of a Real-Time Software System using UML/MARTE, tracing functional requirements into design, ensuring consistency within the developed model, formal verification and also schedulability analysis of RTS using UML/SPT are some contributions presented in this work.

1.4 Issues in Real-Time Software Systems

Several issues are identified related to modeling, consistency, verification and schedulability analysis of RTSS. The issues are summarized as follows.

1.4.1 Modeling

In software engineering practice, especially, for real-time critical systems, defining requirements clearly and unambiguously is not an easy task. As requirements gathering process involves inter-dependency, intercommunication between different groups of people for accurate requirements makes it even more difficult. For example, medical practitioners, who are rarely software engineers, may define medical requirements by using the terminologies which may be difficult to decode by the software engineers. In order to reduce this gap among different groups of people, proper modeling of the actual problem is to be developed first. A model may be developed from different view points and angles by different teams.

Due to non deterministic nature of RTSS, proper modeling is required to implement highly expected results to handle complex situations. Otherwise, it will be expensive for the designers and may result in numerous programming challenges. Specifying and designing of such systems is a complex matter because it requires logical correctness as well as timing correctness. Therefore special attention must be paid to timing during modeling, analysis and verification.

1.4.2 Consistency

Consistency is defined as the state in which any two related design artifacts (in this case UML diagrams) represent the same set of common characteristics or properties of the system. Each design artifact i.e. UML diagram is a set of elements represented by visual notations defined in the standard. UML also provides multi view modeling approach. It consists of a set of diagrams, which gives complimentary views of the same system. Using UML diagrams, structural, behavioral, deployment architecture of the system can be represented. However, these different aspects may be inconsistent.

The word "related" in this context refers to diagrams, which have one or more common elements. This relation is the overlapping of the design views abstracting the same portion of the system. Thus in simple words, a UML based design is consistent in itself if and only if any two UML diagrams, which have some common elements, must have the same value for those elements. In this thesis, RTSS has been modeled using UML/MARTE profile which includes a set of domain models such as timing constraints, general resource allocation. These aspects may contribute to worsen the consistency issue.

1.4.3 Verification

The software development life cycle or in short SDLC incorporates a lot of phases and starting from the traditional waterfall model till the more popular spiral model, every model incorporates some basic phases like – Analysis, Design, Coding and Testing. Boehm defines verification as a process that ensures compliance with requirements specified by the user [14]. Thus verification of requirements in design is defined as a process that ensures that the design is correctly done to implement the stated requirements.

It has been observed that sources of errors can be as early as during the analysis phase and allowing an error to remain undetected gives rise a lot of problems later on. Time and effort required to correct errors detected later is much more compared to those, which are detected earlier. Each requirement must be traced in the design; otherwise, it will be very difficult for developers to identify unimplemented requirements on one hand and managing changes to requirements as well.

The main objective of verification of requirements is to help in the early detection of errors and ensure that system is being built correctly as per user specifications. It also helps in identification of missing requirements, i.e. requirements, which have not been translated to the design phase. This would significantly contribute in decreasing the overall testing effort and time.

1.4.4 Schedulabilty

The scheduler is the part of the operating system that responds to the requests sent by the programs. It interrupts and gives control of the processor to those processes. A scheduler implements an algorithm or policy that determines the order in which processes get a processor for execution. Each task requires the processor to execute; also besides, some tasks may simultaneously need exclusive accesses to one or more of the resources during part or all of their executions. Exclusive accesses to these shared resources are typically ensured only within critical sections. Without proper synchronization, a critical section may be modified by other tasks so that data integrity is violated. The result of this program execution is non deterministic. A resource access-control protocol is a set of rules that govern (1) when and under what conditions each request for a resource is granted and (2) how tasks requiring resources are scheduled [70].

The trouble of giving a guarantee of meeting hard deadlines of tasks lies in the issues of priority inversion and of deadlocks [62]. To beat such difficulties, a great resource access control protocol is required. In RTSS, the main intent of the various resource access control protocols is to schedule and synchronize different tasks when many of these share the same resources.

1.5 Motivation and objectives of work

Developing a real-time software system is a sophisticated and complex task. It is crucial that for such systems, the software should be designed with a robust and sound architecture, which is capable of capturing not only the functional aspects but also the time constraints that are specific to a real-time software system. With the rapid growth in size and complexity of RTSS and to guarantee the development of more reliable systems, modeling plays an important role in the software development life cycle. A minor fault in a model may lead to a major failure of real-time life-saving systems. Development of UML/MARTE based modeling technique is gaining popularity day by day. UML/MARTE model-driven approach can be used to describe non RTSS as well as timing constraints for RTSS. The objective of this research work is to model RTSS using UML/MARTE annotated use case, sequence, timing and class diagrams.

The design modeled in UML/MARTE depicts different aspects, however as they all together represent the same system and there remain some overlapping areas that represent some common characteristics. Hence the diagrams are related with each other and together they seamlessly integrate to represent the system as a whole. So, there are chances of inconsistencies within the models as they depict overlapping characteristics. The objective of this research work lies in proposing comprehensive framework for automatic verification of inter diagram consistency based on UML/-MARTE models.

Requirement traceability helps to verify if all software requirements have been evolved to design, code and test cases. Moreover, verification ensures that each function can be traced to a requirement. With this activity, it is also ensured that all requirements have associated design components and test cases. As real-time systems depend on events under some timing constraints, it is important that the traceability of such must be done completely. The objective of this research work is to facilitate software designers, project managers and architects to automatically generate the traceability metrics of real-time requirements at the early stage of design and to estimate the extent to which requirements have been realized and implemented subsequently in the analysis and design phases. Since changes are more affordable the prior in the development life cycle they are made, this can spare the venture impressive time and cash.

RTS are specially designed to relate to the concurrent behaviour of the real world. Concurrency may lead to resource contention and blocking, which are also important issues to be handled in an RTS. Proper scheduling of the tasks sharing common physical and logical resources is essential to maintain perfect synchronization among tasks. This research work puts stresses on the issues related to deadlock, deadline and time complexity of resource access control protocols. The Priority Inheritance Protocol is used for sharing critical resources but it does not prevent deadlock if nested critical sections are present. This shortcoming is represented using one UML model. Further the Priority Ceiling, Stack Based Priority Ceiling and Stack Based Preemption Ceiling Protocols are used to overcome the difficulty using other improved models.

1.6 Organization of the dissertation

It is evident that new techniques or modification and/or adaptation of existing methodologies are required for RTSS.

Chapter 2 presents review of several research works in the domain of RTSS. Broadly the review presents work that encompasses four domains – Modeling of RTSS, Consistency in RTSS, Verification of RTSS and Schedulability Analysis of RTSS. Several works have been discussed in this area with special emphasis on modeling RTSS using UML. We have surveyed different UML profiles used to model RTSS developed in academia as well as in industry. We have established an assessment of their capabilities and limitations with respect to a variety of criteria such as formal foundations and tool supports etc. Several works focus on tracing requirements into deliverables and artifacts of subsequent phases like design models. The verification of design again is a dominant area of research. Formalization of design specifications and ensuring traceability and consistency within a design are other ways of design verification. We have also focus on the schedulability analysis of RTSS. This chapter discusses some major works in those domains.

Chapter 3 introduces the scope of work presented in this dissertation. We introduce several new strategies, new modeling approaches and methodologies, new analysis techniques and metrics to detect or predict possible errors, inconsistencies or anomalies during analysis and design phases of RTSS.

UML is defined using the metamodeling approach and it has been designed with built-in extensibility mechanisms. These are used to define different domain-specific versions of UML, the UML profile. We have used this approach and leveraged UML extensibility mechanisms to define an extension of UML/MARTE in Chapter 4. This extension enables us to model RTSS using Use case, Sequence, Timing and Class diagrams.

Chapter 5 defines a set of rules to ensure consistency between UML diagrams for the RTSS modeled in Chapter 4. A formal specification of UML diagrams has been introduced for the automated verification of consistency conditions.

Chapter 6 helps in tracing functional requirements into design models and detect possible anomalies like missing requirements or incompatible design not conforming to requirements. This chapter also develops a comprehensive formal verification framework which facilitates automated analysis for ensuring traceability of timing requirements from the requirements analysis phase into the design phase of the software development life cycle.

Chapter 7 compares various resource access control protocols. In this chapter, we have developed a model based on UML/SPT profile to represent deadlock occurrence as a drawback of the Priority Inheritance Protocol. Further, a novel methodology is introduced to prevent deadlock using the Basic Priority Ceiling Protocol, Stack Based Priority Ceiling Protocol and Stack Based Preemption Ceiling Protocol.

Chapter 8 concludes with the advantages of the different modeling, analysis and verification approaches in connection to RTSS presented in the various chapters of the dissertation.

Chapter 2

Review of Related Works

2.1 Introduction

This chapter presents a broad review of research works pertaining to the domain of RTSS. The different methodologies and frameworks discussed here are applied in the different phases of the development life cycle. There are works, which address modeling based on requirement analysis, and several more which address the issues of formalizing real-time modeling based software systems. Starting from analysis and design there are several works that address verification based on functional specifications.

The different research results have been discussed in the order of their applicability in the SDLC. Special emphasis has been given to requirements traceability, the extension of UML mechanism and their representation in RTSS, the use of formal techniques for verification of traceability and consistency, analysis of different resource access control protocols for RTSS.

2.2 Review of UML

UML [87] is a visual modeling language for visualizing, specifying, constructing and documenting the artifacts of software systems. UML [83] is widely used standards for modeling and designing industrial software systems, essentially because it is a semi-formal notation, relatively easy to use and well supported by tools. Originally, UML is the unification of OMT [108], Booch method [15] and Object-Oriented Software Engineering (OOSE) method [57]. This brought the version 0.9 of the UML language. It has been adopted as a standard by the OMG [2] in 1997.

UML consists of a set of diagrams, which gives complimentary views of the same system. Some of the key UML diagrams mostly used are - Use case, Object, Class, sequence, collaboration, state charts, activity, component and deployment diagrams. Much of the research work is now focused on trying to verify the UML model of the software and these works pertain to analysis and verification of one or more UML diagrams. However, UML being a pictorial design of the system, it fails to capture all the aspects of functional needs and hence it is quite difficult to verify systems based on UML diagrams alone. This has given rise to the need for the development of some formal or semi-formal language, which would augment UML and would be able to describe the entire system in totality. Meta Object Facility [86] is one kind of abstract language used for specification of metamodels. OCL [92] on the other hand is a part of UML (1.1 onwards) and a specification language used in conjunction with UML models. It is text based formal language, which allows completely general constraints to be written for the elements appearing in UML models [1]. However, there are several shortcomings of OCL too. Another alternate language for object modeling named Alloy has been proposed in [132] and comparisons between OCL and Alloy have been effectively elicited using examples with emphasis on the shortcomings of OCL and how Alloy manages these issues.

UML provides a variety of instruments to describe the characteristics of a generic system in corresponding models. However, it is not complete, in the sense that the basic elements of the language cannot cover all potential needs for describing specific systems from any domain. Hence in some cases, the definition of domain-specific variants of the UML may be required. The UML, however, has already been conceived for extensibility, for which purpose it provides a built-in extensibility mechanism to extend the language with elements and constructs apt to describe specialized features, though remaining compliant with its standard definition [119]. The extensibility mechanisms in UML are- Stereotypes, Tagged values and Constraints.

UML is not adequate to model domain-specific constructs. In order to fill up this gap, OMG has introduced a mechanism called profiles which have the capabilities to extend the UML metamodel for different purposes. New semantics have also been provided by this extension.

2.2.1 UML profiles

A UML profile is a special version of UML tailored to the specifics of a particular domain, like the real-time domain, or a particular activity, like system requirement modeling. There are several UML profiles in the literature. In the following section, we have considered some of the UML profiles.

2.2.1.1 UML Profile for Schedulability, Performance and Time

The UML profile for real-time modeling formally called the UML profile for Schedulability, Performance and Time (UML/SPT), was adopted by the OMG in 2002 [85]. UML/SPT is a framework to enables the modeling of resources and quality of service; time concept and time related mechanisms; and concurrency. This increased the interest in the use of object-oriented technology and UML, in particular, to model and build real-time systems [113]. It provides the user (modeler) with a set of stereotypes and tagged values to annotate the UML models. In addition, UML/SPT supports Quantitative analysis (schedulability and performance analysis) which enables the prediction of key properties in the early stages of a software development process. The structure of the UML/SPT, as illustrated in Figure 2.1, is composed of a number of sub-profiles:

- The General Resource Model (GRM) package is the core of UML/SPT. It is further partitioned into three packages:
  - RT resource Modeling for the basic concepts of quality of service and resource
  - RT Concurrency Modeling for concurrency modeling
  - RT time Modeling for time and time-related mechanisms modeling
- The Analysis Modeling package defines the analysis sub-profiles, including:
  - PA profile for performance analysis
  - SA profile for schedulability analysis

2.2.1.2 UML Profile for Quality of Service

The UML profile for modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (UML/QoS) was adopted by the OMG in 2004 [88]. It is an emerging UML profile that aims at capturing the concept of quality of service at large. It allows the definition of an open variety of quality of service requirements and properties [28]. UML/QoS is relevant for real-time software modeling because it is defined to complement the aforementioned UML/SPT. However, while UML/SPT is tailored to fit performance and schedulability analysis, UML/QoS allows the designer to define any set of quality of service requirements and carry out any specific analysis that could be relevant for the safety-critical aspect of real-time software. This was demonstrated in [11], where a quality model has been defined to drive a

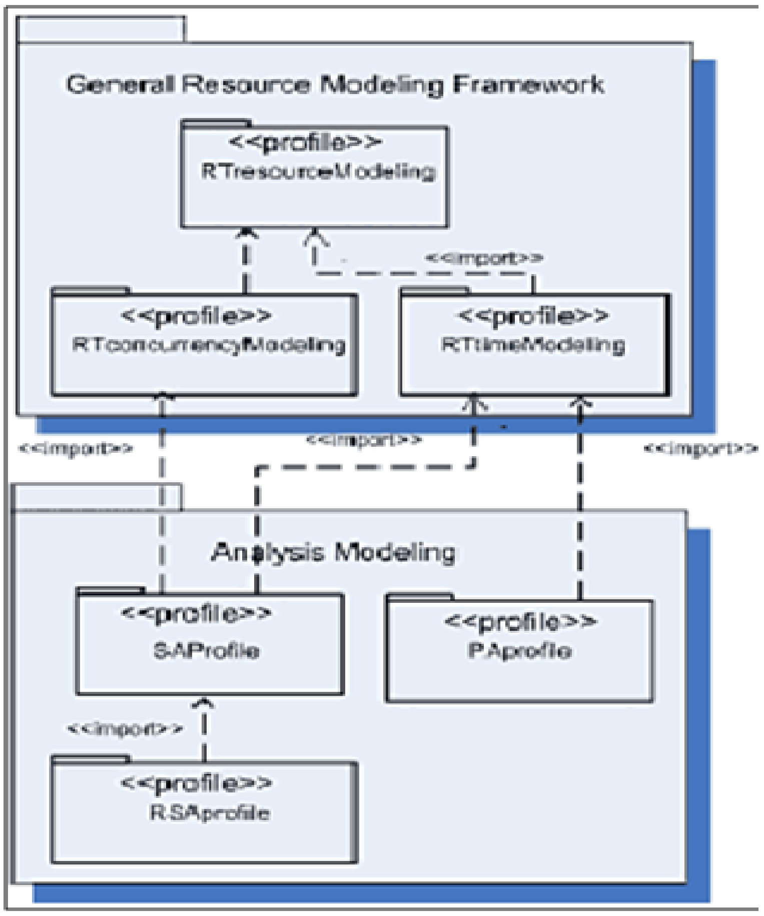


FIGURE 2.1: Structure of UML/SPT

dependability and performability analysis of an embedded automation system. In a nutshell, UML/QoS could also be used to annotate UML diagrams. In contrast to UML/SPT, UML/QoS proposes a procedure that consists of three steps:

- Definition of QoS characteristics: The new user-defined QoS characteristics could leverage, through specialization, the general QoS characteristics catalog defined in this profile. This catalog comprises the following categories: Performance, Dependability, Security, Integrity, Coherence, Throughput, Latency, Efficiency, Demand, Reliability and Availability. In particular, the Latency

QoS characteristics can be used in the real-time context. The QoS characteristics are templates classes having parameters. The latter has to be instantiated in the next step.

- Definition of the quality model: The QoS characteristics parameters should be assigned actual values. This is done through the definition of quality characteristics bound class and template bindings. The UML model containing the binding information and the bound classes is called the Quality Model.
- The last step is the UML models annotation using quality of service requirements.

2.2.1.3 UML-RT Profile

Rational Software has developed a real-time profile named as UML-RT [116]. Using the UML built-in extensibility mechanisms, UML-RT captures the concepts of ROOM [118]. UML-RT allows the designer to produce models of complex, event-driven and possibly distributed real-time systems. However, it does not support time and timing constraints modeling. UML-RT is supported by a CASE tool called RationalRT that allows for automatic code generation by compiling the models and linking them with a run-time system. UML-RT includes constructs to model the structure and the behavior of real-time systems:

- Structure Modeling: UML-RT provides the designer with entities called capsules, which are communicating active objects. The capsules interact by sending and receiving messages through interfaces called ports. Furthermore, a capsule may have an internal structure composed of other communicating capsules and so on. This hierarchical decomposition allows the modeling of complex systems.
- Behavior Modeling: The behavior is modeled by an extended finite state machine, and it is visualized using UML state diagrams. These state machines are

hierarchical since a state could be decomposed into other finite state machines. A message reception triggers a transition in the state machine. Actions may be associated with transitions or the entry and/or the exit of a state. Similar to the two previous UML profiles, UML-RT lacks formal foundations. UML-RT is, however, a basis for a very active research work on schedulability analysis applied to real-time software design models. Indeed, while the CASE tool RationalRT allows an automatic code generation, it does not take into account timing constraints. Therefore, the research reported in [114] was a first attempt to integrate the real-time schedulability theory with object-oriented design targeting real-time systems.

2.2.1.4 UML/MARTE Profile

The UML profile for MARTE [94] was defined to provide several some many concepts that modelers can use to express relevant properties of Real-Time Embedded Systems, for example, related to performance and schedulability. Shortcomings of UML/SPT [85] profile come in terms of its expressive power and flexibility. For example, it was necessary to support the design of both hardware and software aspects of embedded systems and a more extensive support for schedulability and performance analysis, encompassing additional techniques such as hierarchical scheduling.

The objective was to address the above issues as well as to provide alignment with the UML profile for Quality of Service and Fault Tolerance [88], which enables specification of not only real-time constraints but also other embedded systems characteristics, such as memory capacity and power consumption. MARTE was also required to support modeling and analysis of component-based architectures, as well as a variety of different computational paradigms (asynchronous, synchronous, and timed). MARTE profile is used for strengthening the expressive power of UML. It supports modeling and analysis in RTES. At present, MARTE 1.1 [90] is released formally. MARTE provides predefined stereotypes and tagged values to specify

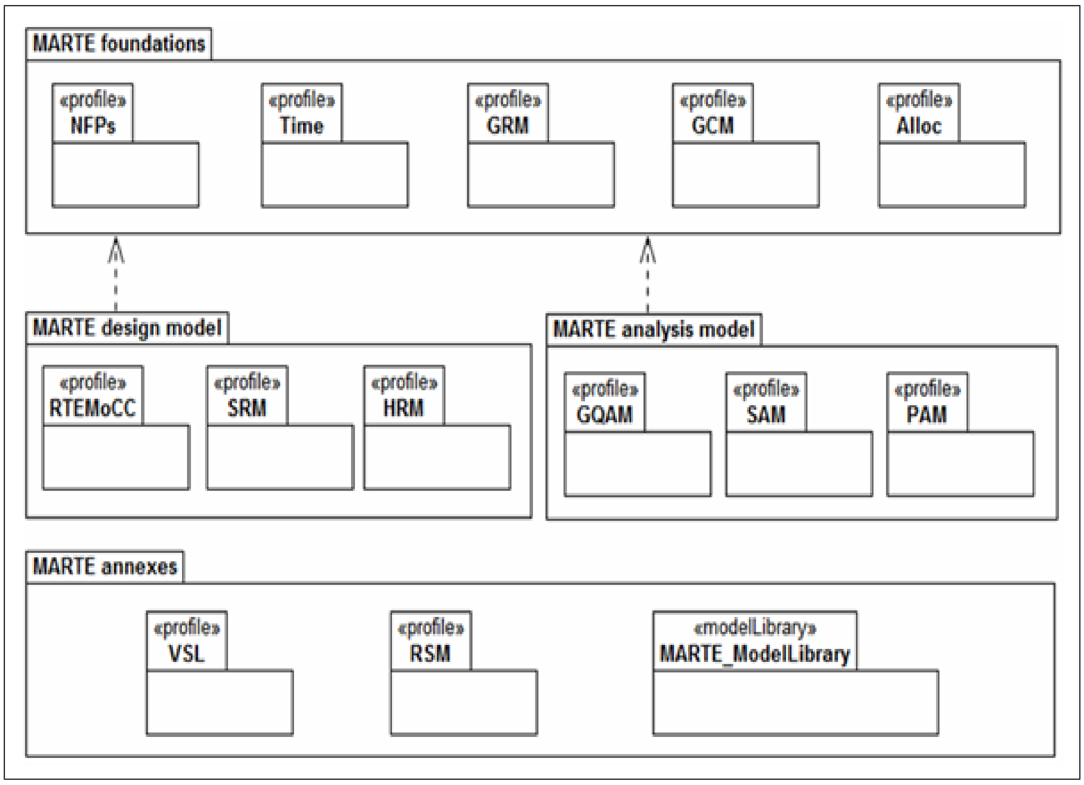


Figure 2.2: Structure of UML/MARTE

non-functional properties, time related constraints, general resources (software and hardware). Those modeling artifacts are essential concepts to design real-time software systems.

MARTE is organized into four main packages to represent different concepts of a RTES at different levels of abstraction of the real system. Structure of UML/-MARTE is shown in Figure 2.2.

Four packages are MARTE foundations, MARTE design model, MARTE analysis model and MARTE annexes.

MARTE foundations define concepts for real-time and embedded systems. It includes the following packages

- Non-Functional Property (NFP) package provides a general framework for annotating UML profile for MARTE. This provides modeling constructs for declaring, qualifying and applying semantically well-formed non-functional aspects of UML models. It is complemented by the Value Specification Language (VSL), which is a textual language for specifying algebraic expressions. The NFP sub-profile supports the declaration of non-functional properties as UML data types, whereas VSL is used to specify the values of those types and their potential functional relationships.

- The Time packages provides a general framework for representing time and time related concepts and mechanisms that are relevant for modeling real-time and embedded systems.
- The Generic Resource Modeling (GRM) package represents the set of resources underlying an application and also how the system uses them. It consists of an ontology of resources enabling modeling of common computing platforms (i.e., a set of resources on top of which an application may be allocated to be computed), and high level concepts for specifying resource usage. The level of abstraction used here is at a general system level.
- Allocation modeling (Alloc) package provides some general concepts pertaining to the allocation of functionality to entities responsible for its realization. It may be either time-related allocation (i.e., scheduling) or space allocation.

MARTE design model provides concepts required from specification to detailed design of real-time embedded systems such as generic component model (GCM), high-level application modeling (HLAM), software resource modeling (SRM) and hardware resource modeling (HRM).

- The Generic Component Model (GCM) package defines the concepts necessary to address the modeling of artifacts in the context of real-time and embedded systems based component approaches. It enables execution platform modeling and provides the foundations needed for more refined modeling of both hardware and software resources. GCM supports both message and data based communication schemes between components.

- Model-based design of RTE systems with MARTE proceeds mostly in a declarative way. Hence, MARTE users may annotate their models with real-time concerns using the extensions defined within the HLAM (High-Level Application Modeling) package.
- SRM is used to create a platform independent design which enables to create a general model to accommodate all the different types of a real-time operating system in use.
- HRM is used to express hardware entities by providing several stereotypes through three different views:
  - a high-level architectural view
  - a specialized view
  - a detailed physical view

Model-based analysis using MARTE is done using mainly the extensions defined either in the Generic Quantitative Analysis Modeling profile (GQAM) or using one of its two refinements, dedicated respectively to schedulability analysis (SAM) and performance analysis (PAM). The annotation mechanism used in MARTE to support model-based analyses uses UML stereotypes. These typically map the UML model elements of the application into corresponding analysis domain concepts and also allow specification of values for properties which are needed to carry out the analyses.

MARTE annexes provide a predefined MARTE model library and value specification languages such as the Value Specification Language (VSL), the Clocked Valued Specification Language (CVSL), and the Clock Constraint Specification Language (CCSL).

2.2.1.5 TURTLE Profile

TURTLE stands for Timed UML and RT-LOTOS Environment. It is a UML profile aiming at the formal validation of complex real-time systems [6]. TURTLE uses UML's extensibility mechanisms to enhance UML structuring and behavioral expressive power. It has a strong formal foundation. TURTLE extensions semantics is expressed by mapping to RT-LOTOS. This enables a formal validation as well as a simulation of the UML models. TURTLE essentially allows the description of the structure/architecture as well as the behavior of the system using an extension of the UML class, object and activity diagrams.

The main extensions brought by TURTLE are the following:

- Structural Extensions: TURTLE introduces the concept of TClass, which has special attributes called Gates. These are used by TClass instances, TInstances, to communicate and are specialized in InGate and OutGate. In addition, TURTLE introduces stereotypes called composition operators. These are used to explicitly express parallelism, synchronization, and sequence relationships between TClasses.
- Behavioral Extensions: The behavior of a TClass is expressed using activity diagrams extended with logical and temporal operators. These operators allow expressing synchronization on gates with data exchange. Moreover, TURTLE enables the expression of temporal non-determinism and different sorts of delays (deterministic, non deterministic).

TURTLE is supported by a toolkit composed of RTL [107] and TTool [130]. These are used by the designer to build a design, to run the simulation and to perform a reachability analysis for the validation of the system. Finally, TURTLE was extended to fit the requirements of distributed and critical systems. The objective is to enable the definition of components and their deployment, and to study their properties at early stages of the software development process. This is done using a formal definition of the deployment diagrams, which are the most suitable for distributed architecture description. Therefore, TURTLE has been extended to take into account deployment diagrams. The obtained profile is called TURTLE-P [7], which addresses the concrete description of communication architectures. TURTLE-P allows the formal validation of the components and deployment diagrams through its foundations in RT-LOTOS.

2.2.1.6 SDL combined with UML

This is the title of the ITU-T recommendation Z.109 [56, 78]. It is a UML profile for SDL since it defines a specialization of a subset of UML and a one-to-one mapping to a subset of SDL. Thus, Z.109 has SDL as a formal semantics. This profile provides the designer with a combination of UML and SDL. Essentially, Z.109 defines a UML model for the main concepts of SDL, the domain model and offers a corresponding set of stereotype. In the following, we highlight the main concepts defined in Z.109.

- Agent: An SDL system is composed of agents connected through channels. An agent has a state machine and an internal structure composed hierarchically of other agents. Moreover, an agent can be a process, a bloc or a system. In particular, an agent type is mapped into a class of active objects and its kind is stereotyped <<system>>, <<block>> or <<process>>.
- Gates and Interface: The agents communicate through gates by sending signals or requesting a procedure, which together, the signals and procedures, compose its interface. The latter is mapped into a UML interface and the former are stereotyped <<signal>> and <<procedure>>.
- State Machine: An SDL agent state machine is mapped to a UML state machine.
- Package: UML packages are used to represent SDL packages.

Finally, this profile has been implemented in the Telelogic CASE tool Telelogic TAU 3.5 [129].

2.2.1.7 OMEGA UML Profile

This profile [48] compliant is part of the OMEGA project [125]. It is a framework for UML-based real-time modeling allowing for the analysis and verification of time and scheduling aspects. It provides a set of timed-events primitives and the semantics of these primitives is expressed formally in terms of timed automata with urgency.

2.2.1.8 OCL Profile

This profile is based on an extension of OCL 2.0 metamodel [41]. It allows for the specification of real-time constraints using OCL. The formal semantics of this profile is given by a mapping to time-annotated temporal logic formulae expressed in CTL. This enables formal verification of different system properties.

2.2.1.9 UML Profile for System Engineering

The Systems Modeling Language (SysML) is a general-purpose modeling language for systems engineering applications. It supports the specification, analysis, design, verification and validation of a broad range of complex systems. SysML was originally developed by an open source specification project and includes an open source license for distribution and use. SysML is defined as an extension of a subset of UML using UML's profile mechanism.

The SysML initiative originated in a January 2001 decision by the International Council on Systems Engineering (INCOSE) [54] Model-Driven Systems Design workgroup to customize the UML for systems engineering applications. Taxonomy of SysML diagrams is shown in Figure 2.3.

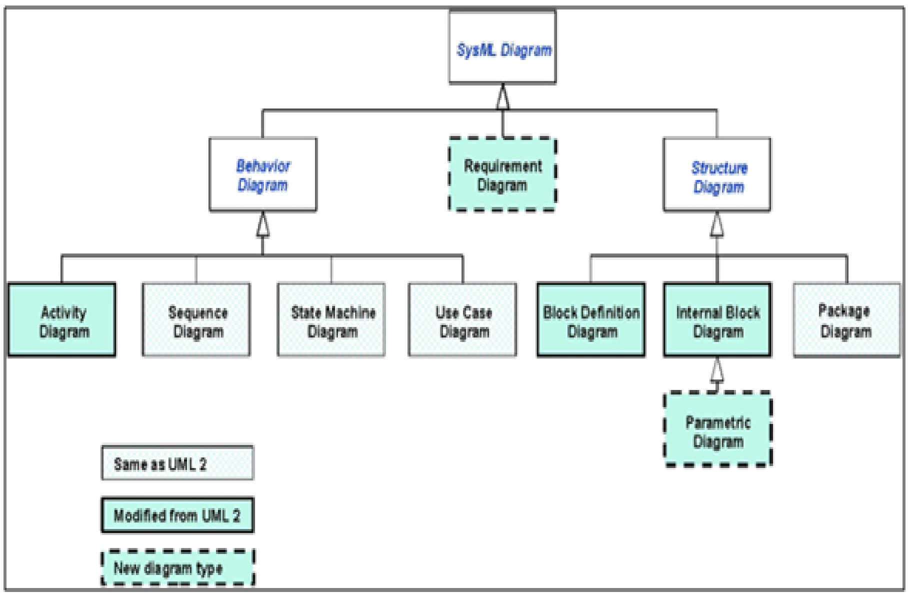


FIGURE 2.3: Taxonomy of SysML diagrams

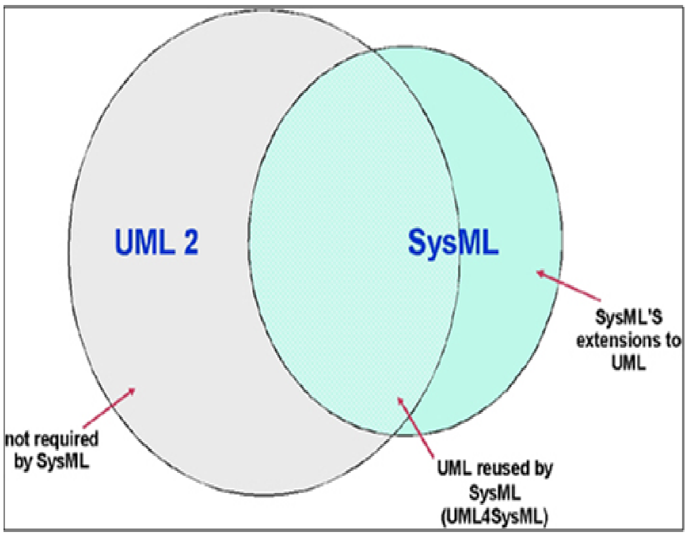


FIGURE 2.4: SysML Relationship

SysML reuses a subset of UML 2.1 and extends it by additional diagrams and new concepts as it is shown in Figure 2.4. SysML offers systems engineers several noteworthy improvements over UML, which tends to be software-centric. These improvements include the following:

- SysML's semantics are more flexible and expressive. SysML reduces UML's software-centric restrictions and adds two new diagram types- requirement and parametric diagrams. The former can be used for requirements engineering; the latter can be used for performance analysis and quantitative analysis. Consequent to these enhancements, SysML can model a wide range of systems, which may include hardware, software, information, processes, personnel, and facilities [93].
- UML does not allow to represent the trace of the informal requirements specification to the system design elements. Generally, UML Use Cases are used to understand the expected system functionalities but the requirements are traced to the use cases and not to the design. With this regard, SysML brings a major enhancement through the requirement diagrams. These allow representing the requirements and many relationships among them as well as their relationship with the system architecture and design elements.
- SysML allocation tables support common kinds of allocations. Whereas UML provides only limited support for tabular notations, SysML furnishes flexible allocation tables that support requirements allocation, functional allocation, and structural allocation. This capability facilitates automated verification and validation (V&V) and gap analysis.
- Another important feature of SysML is the introduction of additional models of computation. It extends the behavior of UML activity diagrams so that the control of execution of a running action can be disabled. It is also extended to enable the modeling of continuous and probabilistic systems.

- SysML model management constructs support models, views, and viewpoints. These constructs extend UML's capabilities and are architecturally aligned with IEEE-Std-1471-2000 (IEEE Recommended Practice for Architectural Description of Software-Intensive Systems).

2.2.1.10 UML Profile for Systems-On-Chip

System-On-Chip refers to the integration of computing and communications components into a single chip. SoCs are incorporating more processors and software. The success of UML in the software community has led to a surge of interest in using UML in the SoC design flow [76]. In this context, the focus is put on how to customize UML so that it can be used as System Level modeling language in the SoC design flow. Therefore, many of these customizations are UML profile for SystemC [128] which is a C++ based system level language used in the SoC design flow [80]. The OMG standardized profile for SoC [89] is defined to support the modeling and specification of SoC designs. In particular, it introduces SoC structure diagrams. In addition, the profile defines a set of stereotypes to represent modules, connectors, ports, channels, clocks, processors, protocols, data types and connectors.

We have studied different UML Profiles for real-time and drawn a comparison between the aforementioned profiles according to some criteria, including formal foundation, expressiveness and tool support [45]. This comparison is summarized in Table 2.1.

2.3 Modeling of RTSS

UML is comparatively easy to use and it has enough tool support to describe the artifacts of software systems. UML has "real-time capabilities" [32, 85] and it has been used in many resource-critical and time-critical systems. But UML also has

TABLE 2.1: UML Profiles for Real-Time Systems

	Issuer	**Tool Support**	**System**	**Analysis**	**Expressiveness**	**Formal Semantics**
UML/SPT	Industry (OMG)	Rhapsody (iLogix) RT Studio (Artisan)	Generic and real-time systems	Performance Schedulability	Time, Resource Concurrency	No
UML/QoS	Industry (OMG)	None	Generic and Real-Time Systems	User defined	QoS	No
UML-RT	Industry IBM/Rational	Rational RT	Event driven Real-Time Systems	Schedulability	Capsules, port Async mesages	No
TURTLE	Academia	TTool RTL	Distributed Critical Systems	No	Synch, Parallel delays operators	RT-LOTOS
Z.109	Industry ITU-T	Telelogic Tau 3.5	Telecommunication Critical Systems	No	SDL concepts	SDL
OMEGA-RT	Academia	OMEGA tool set	Real-Time Systems	Timing scheduling	Timed events	Timed automata with urgency
OCL	Academia	None	Generic and Real-Time Systems	Verification	Real-time constraints	CTL

few limitations. It cannot describe a system from all possible domains through its basic elements and thus it cannot be considered as a complete language for system modeling and it does not support some important characteristics of RTS such as modeling of timing constraints, signals and independent components [12] etc. Lavazza et.al. [64] have proposed an approach for automatic translation of UML models into a formal representation that support the verification of properties, for example, safety, utility, liveness etc.

The RTSS pose a unique set of challenges with different kinds of timing constraints such as duration, delay and deadline associated with events and schedulability concerns. In [116], the authors describe a set of constructs that facilitate the design of software architectures in the domain of real-time software systems using the UML and utilizing the power of its extensibility mechanisms.

UML is suited for the analysis of RTS and this forms the basis of the review of some of the important UML profiles relevant for RTS in [45]. Formal semantics of UML-RT have been introduced in [100]. As real-time scheduling analysis can not be performed by using UML-RT, the authors in [49] provide a solution for real-time schedulability analysis using the RoseRt case tool. [9] offered a sub-profile of SPT for schedulability modeling.

In [23] Z.Chen et al. have introduced the semantic description of Refinement of Component and Object Systems (rCOS). Using the concept of this model, they proved how it could integrate the model-driven approach. In [121] a general approach is proposed which detects concurrency problems using a genetic algorithm. This is achieved by using UML/MARTE profile. In [55] Iqbal et al. have introduced a framework for practitioners to reduce the gap between the modeling notations and real-world industrial applications using UML/MARTE.

In [104, 109] formal semantics for automated translation of Timed Rebeca has been proposed. The authors in [127] propose a language called Stateful Timed CSP and an automated approach for checking the model. However, we have developed a framework on UML/MARTE, which is a more popular modeling tool for software systems and hence would be more useful for analysis of RTSS. In [139] Gregory et al. introduced a UML profile called SafeUML in the context of RTCA DO-178B to improve the communication between safety and software engineers through an automated generation of certification-related information.

Researchers found fruitful ways of combining Formal Verification Tools (FVT) with CASE tools. This approach made sensible progress on the interpretation of graphical descriptions to model checkers' input symbols. However, these mechanisms were unable to return the results of verification into the CASE tool's process. Mota et al. [79] proposed a protocol interface that merges both technologies and tries to overcome the drawback of the above mentioned mechanisms. The Primary focus of [72] is the conversion of HMSC (High-level MSC) semantics to timed automata.

In [31], Douglass has used the patient ventilation system as a case study to demonstrate how well UML can be used to represent a real-time system development problem. But, the work lacks several important system specifications, parameters, and time-based constraints for modeling RTS. Further, the absence of formal approach made it difficult for the automated analysis. We have extended the MARTE metamodel to introduce new user defined data types required to model a real-time software system.

2.4 Consistency in RTSS

Consistency is a state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe, which are jointly satisfiable [123]. It is one of the attributes used in measuring the quality of the Unified Modeling Language (UML) model [82]. According to [52, 123] there are three main activities in model consistency management. They are consistency specification, inconsistency detection and inconsistency handling. Consistency rules, which must be represented by different diagrams for them to be consistent, are specified first. If the consistency rules are not fulfilled, inconsistencies were aroused and they should be detected and handled. Even though there is increasing research in consistency between diagrams as reviewed by [71], there is still lack of researches of consistency driven by Use Case. In famous system development methodologies such as ICONIX [106] and Rational Unified Process (RUP) [53], Use Cases provide the foundation for defining functional requirements and design throughout system development. The importance of Use Case can be seen in [30] as it is second ranked diagram used by UML practitioners.

A novel approach is introduced in [65] to deal with consistency verification between a unique software system and its advancement at both design and implementation phases. An algorithmic approach to a consistency check between UML Sequence and State diagrams is described in [69], while [61] proposes a declarative approach using process algebra CSP for consistency checking between sequence and statecharts. SPIN model checker has been used in [138] to verify the consistency between the sequence and statechart diagrams. A tool is developed to verify the consistency issue automatically. In [34] an approach for automated consistency checking named VIEWINTEGRA has been developed. It provides support for active (preventive) and passive (detective) consistency checking. A consistency concept is presented in [60] that focuses on the establishment of timing constraints. Similarly, the authors in [46] present a consistency framework for UML/SPT models. Research work in [112] focuses on the different types of inconsistencies and the consistency rules present in the design phase of the software development life cycle. Shinkawa [120], Sapna et al. [115] and Chanda et al. [20] propose consistency between use case and activity diagrams. Shinkawa [120] specifies consistency between use case, activity, sequence and state chart diagram using Colored Petri Net (CPN). He proposes that a use case may have at least an activity diagram. He also defines the use case, action and execution occurrences as transitions.

While Sapna et al. in [115] defined elements of use case, activity and sequence diagrams using schema table. But the definitions are just limited to elements of use case, actor, activity, message and object. A use case may have an activity diagram, each actor in use case diagram is matched to a class in the activity diagram. They define that each object and its messages in the sequence diagram correspond to a class and its methods in the class diagram. They proposed two (2) consistency rules between use case and sequence diagrams.

Our framework in comparison is formal and hence facilitates automated analysis for identification of inconsistencies arising between UML diagrams for RTSS. Chanda et al. [20] express elements of use case, activity and sequence diagram as context-free grammar. They have defined relationships among diagrams. The formal syntax of each diagram is then used to reason the rules using CFG. However, they have worked on non-RTS and do not consider any real-time requirements such as timing constraints. Elements defined in consistency rules by [20, 115] do not follow abstract syntax standardized by OMG [84].

Li et al. [67] have proposed formal semantics of the UML sequence diagram to catch the consistency between the sequence diagram with the class diagram and state diagram. This methodology might be helpful to build up the model consistent checking functions in UML CASE tools and furthermore to reason about the accuracy of a design model regarding a requirement model. With respect to timing constraints in the sequence diagram, Li et al. [68] described an algorithm based on linear programming that analyzes whether several timing constraints within a sequence diagram are consistent with each other. They extend their approach to compositions of sequence diagrams. Fryz et al. [43] consider a use case diagram as user requirements and they have described the diagram as a graph. They have defined consistency between use case and class diagram using conjugated graphs.

In [21], Chechik et al. describe methods and tools for automatically analyzing the consistency of software requirements and detailed designs. A tool is implemented which checks this notion of consistency is satisfied. Their related research works are present in [33, 110, 111]. In [36], Egyed has introduced an automated approach for consistency detection. If model changes, the proposed approach can detect and track inconsistencies in real-time automatically. In [24], Jinho et al. have proposed a systematic approach for checking timing consistency rules among three UML diagrams - state machine, sequence and timing diagrams using two case studies with MARTE annotations. However, their approach lacks formal verification.

Authors in [31] model real-time patient ventilation system, but this work does not provide any consistency checking mechanisms among various UML diagrams. In comparison to that, we have defined the consistency rules among various UML diagrams and formally verify these. Contribution of Chapter 5 lies in defining a set of consistency conditions between related UML/MARTE models. Further, a formal approach has been developed for the automated verification of the consistency rules and details of this new approach have been presented in that chapter.

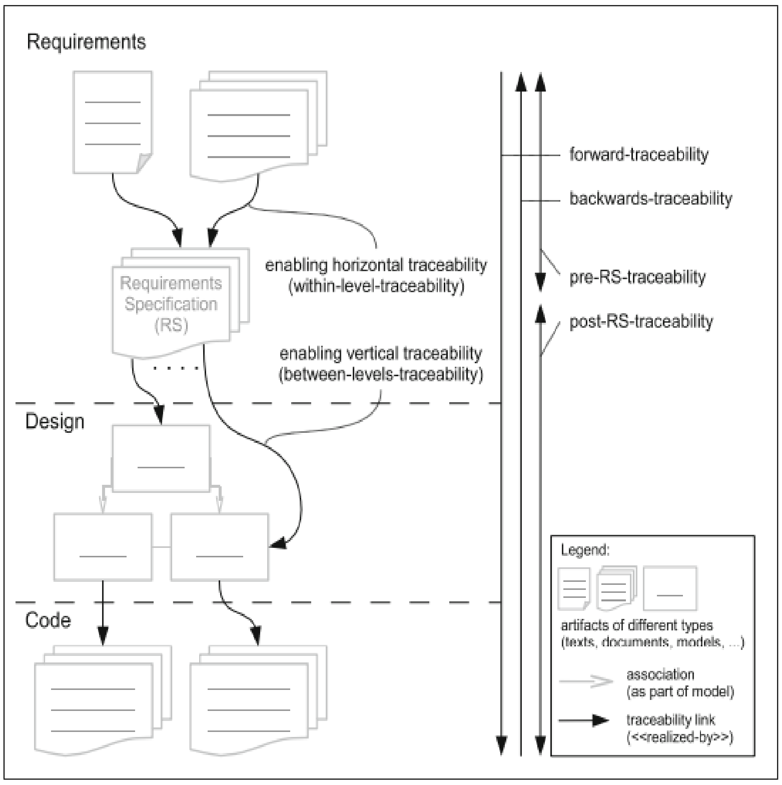

Requirements
Requirements Specification (RS)
enabling horizontal traceability (within-level-traceability)
enabling vertical traceability (between-levels-traceability)
Design
Code
forward-traceability
backwards-traceability
pre-RS-traceability
post-RS-traceability
Legend:
artifacts of different types (texts, documents, models, ...)
association (as part of model)
traceability link (<<realized-by>>)

FIGURE 2.5: Different types of Traceability Link

2.5 Verification of RTSS

Requirements traceability can be defined as "the ability to describe and follow the life of a requirement, in both a forward and backward direction" [47]. The general consensus is that traceability between different artifacts helps reduce development and maintenance time and cost thereby improving the quality of the system [3]. Software traceability refers to the process of discovering and maintaining links between the different artifacts. Different types of traceability links are pre requirements specification (pre-RS), pre-requirements specification (post-RS), forwards, backwards, horizontal, and vertical traceability. These are shown in Figure 2.5 [134].

Authors in [73] proposed that traceability not only reduces downstream cost but can also improve software maintenance quality. Utilizing the metamodeling approach, authors in[81] propose requirements modeling which permits the reconciliation of the expressiveness of a portion of the more pertinent Requirements Engineering systems. This research work concentrates on versatility concerning expressiveness and flexibility to the application area to set up some essential rules and expansion components that advance knowledge and semantic exactness. In [102], reference models including the most imperative sorts of traceability links for different improvement tasks have been orchestrated using an empirical approach.

Alexander Egyed in [35] introduces a new, strongly iterative approach to trace analysis, which will automatically generate new trace dependencies and validate existing ones between model elements, scenarios and code. The proposed approach is fully automated and has been supported by case studies. A tool is introduced in [10] to permit the mix of this semi-formal development method with a formal strategy to empower framework confirmation without any knowledge of either formal languages or temporal logic.

In [37] authors develop a system based on existing Requirement traceability research. With the help of software attributes, the proposed system can find out requirements conflicts. Using a trace analysis technique, automatically discards false conflicts and cooperation. Using heuristic traceability rules, the proposed system in [122] automatically generates traceability relation between textual requirement artifacts and object models.

Automatic traceability rules between requirements and analysis documents have been proposed in [124] using a rule-based technique. The work in [134] aims to bring software developers, requirements engineers and model-driven developers together by providing an overview of the current state of traceability research and practice in both areas. The concept of event-based traceability is discussed in [25] which is a new method of traceability based upon event-notification and is applicable even in a heterogeneous and globally distributed development environment.

In order to support the software maintenance process, authors in [137] have proposed a novel approach to set up the traceability links between the existing source code and document by formally creating ontological representations for both the artifacts. Authors in [50] concentrate on enhancing the recall and precision to decrease the quantity of missed traceability links and in addition to lessen the quantity of unimportant potential links that an expert needs to inspect when performing requirement tracing. Due to software advancement, or to the absence of a taught improvement and support process, traceability links regularly have a tendency to be obsolete or missing. To help the prerequisites investigator in the choice about when to quit assessing candidate traceability links, authors in [39] propose a novel approach called Estimating the Number of Remaining Links (ENRL) which goes for evaluating the quantity of remaining positive links in a ranked list of candidate traceability links created by a Natural Language Processing methods.

Traceability metrics are usually constructed manually in many software development industries. Hence, it is costly to maintain them. To address this issue authors in [26], [5] have proposed an approach to investigate the use of dynamic retrieval methods to automatically generate the traceability links. Candidate link generation is one of the important part of the requirement tracing process. Research work in [51] provides the various new measures to improve the quality of the candidate link generation.

In [66], authors proposed a formal software verification technique using equality loop invariants. In order to verify the traceability of non-functional requirements, authors propose a design based methodology using EAST-ADL2 and MARTE languages [96] for safety or mission-critical systems. Authors in [75] exhibit a model-driven approach for requirement engineering by integrating UML, MARTE, and SysML models to enhance and encourage requirements specification. To automate the proposed approach, authors build up a tool which consequently creates traceability metrics and permits recognize possible inconsistencies when changes in the requirements are made.

Authors in [105] combine the application of SysML with MARTE stereotypes to specify the important elements of individual software requirements for RTS and furthermore represent traceability between requirements. In any case, the work does not verify the traceability between requirement and design phase of the software development life cycle and furthermore does not provide any formal specification to check the non-functional requirements in the outline. In contrast with that, this research work formally verifies the requirements traceability of timing constraints between the requirements analysis and design phase of a software development life cycle. The fundamental commitment of [58] is to develop a set of metrics based on use case, sequence and class diagrams. However, authors have worked on non-RTS and does not consider any real-time requirements such as timing constraints. In comparison to that, our work further enhances the concept of RTSS and shows how requirements related to timing constraints are traced automatically from the use case diagram through the sequence diagram to the timing diagram and finally to the class diagram.

In order to verify and validate an automated system, a model checking approach can be utilized. Research work in [40] proposes a framework for the translation of use case, activity and state diagrams to symbolic model checker. ATL model checker is implemented with the help of an algebraic approach in [126]. ANTLR has been used to implement the algebraic compiler included within ATL model checker. Related research works are also provided in [18]. A similar approach is also used to implement CTL model checker [17]. The existing model checking tools (Spin, CADP, Alloy, FDR2 etc.) [42] require input specifications to be provided in some specific manner, like Promela for Spin, first order logic for Alloy, LOTOS-NT for CADP and CSPm for FDR2 etc. In contrast with that, our framework minimizes this extra overhead. Given a UML model, the framework automatically verifies the requirements traces and generates the traceability metrics.

In this dissertation, we present a metric based verification on a formal framework which focuses on tracing requirements into design artifacts. With the help of semantic actions, the formal specification automatically derives trace metrics to quantitatively assess the extent of real-time requirement coverage in the system. Our approach discussed in detail in Chapter 6 focuses on the identification of those aspects.

2.6 Schedulability Analysis of RTSS

L. Sha et al. [119] have proposed uniprocessor resource sharing schemes, the basic Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) to overcome the difficulties of the uncontrolled priority inversion problem. These protocols provide a well-known method for fixed priority scheduling systems. The PCP is a refinement of the PIP. The PCP prevents deadlocks and reduces blocking time compared to the PIP. T. P. Baker [8] has developed a uniprocessor resource sharing algorithm i.e. the Stack-Based Resource Sharing Protocol (SRP) for the Earliest Deadline First (EDF) scheduling and the Rate Monotonic Scheduling. The SRP manages the execution of the tasks according to their preemption levels which are assigned statically based on their priorities and release times. This protocol is the refinement of the concept of priority ceiling [101, 119]. A concurrency control protocol to specify the dynamic priority ceiling of the tasks for the EDF scheduling has been proposed, which overcomes the drawbacks of the occurrences of the deadlocks and chained blocking [22].

In RTS uncontrolled priority inversion may cause the missing of the deadlines which is undesirable in hard RTS. An alternative protocol, the Immediate Priority Ceiling of the priority inheritance is proposed and implemented in [19] for real-time drivers. Lam et al. [63] have proposed a Priority Ceiling Protocol with Dynamic Adjustment of Serialization Order for real-time database systems. If data conflicts occur, a higher priority transaction preempts a lower priority transaction with the help of the dynamic adjustment of the serialization order. This protocol prevents the occurrences of deadlocks as well as the unnecessary blocking. Maroua Gasmi et al. [44] have proposed Reconfigurable Priority Ceiling Protocol (RPCP). In a random scenario of reconfiguration, the problems of deadlock and missing of deadlines can occur. This protocol finds solutions to the optimization of blocking and response times. Authors have developed a simulation tool at LISI Lab (University of Carthage) which was applied to a case study to show the contributions of their research.

Kiss [59] has proposed the Intelligent Priority Ceiling Protocol (IPCP) where the optimal priorities have been approximated by applying adaptive techniques. This protocol has two major parts. The first part optimizes the priorities of the actual task and the resource set, while the second part tunes the priorities in the system according to the system wide parameters. Xibo Wang et al. [133] have carried out a research based on the layered scheduling algorithm which is a refinement of PIP and PCP. The Layered Priority Inheritance Protocol (LPIP) and the Layered Priority Ceiling Protocol (LPCP) are proposed in a layered scheduling environment where priority inversion and occurrence of deadlock are prevented.

Exact schedulability analysis for EDF scheduling is provided by considering both the resource sharing and release jitter [135]. An extension of the Shared Resource Protocol has been proposed to minimize the blocking time [13]. Authors in [136] present a formalized and mechanically checked verification for the rightness of only priority inheritance protocol. Maria Cruz Valiente et al. [131] have scheduled a set of identified tasks using the capabilities of UML. The PIP is used to share critical resources. But one of the major drawbacks is that it does not prevent deadlock when nested critical sections are used. In this research work, we have developed a new resource access protocol which overcomes this difficulty using an improved model.

Deadline Floor inheritance Protocol [16] is proposed as an alternative of SRP. In this protocol, the relative deadline is relegated to all the shared resources. Deadline of a resource will be same as the minimum (floor) of all the tasks which utilize that resource. Whenever a resource is allocated to a task, the absolute deadline for that task will be minimized to mirror the resource's deadline floor. But, this work posses extra overhead to calculate the relative deadline of all resources as well as to modify the task's deadline whenever a resource is allocated and released.

Taking the cue from the above works, Chapter 7 compares various resource access control protocols. The main objective of this chapter is to model, analyze and verify four existing protocols using UML/SPT based Sequence and Timing diagrams to study deadlock in RTSS.

2.7 Conclusion

This chapter presents a comprehensive review of research works pertaining to the domain of RTSS. We have discussed the works by categorizing them as per their applicability in the RTSS namely – modeling, consistency, verification and schedulability analysis. In the context of the existing review of research works presented here, we develop an integrated approach in this dissertation, which forward towards modeling, consistency, verification and schedulability analysis of RTSS. The next chapter gives an overview of the scope of work.

Chapter 3

Scope of Work

3.1 Introduction

Developing a real-time software system is a sophisticated and complex task. It is crucial that for such systems, the software should be designed with a robust and sound architecture, which is capable of capturing not only the functional aspects but also the time constraints that are specific to a real-time software system. This chapter provides a brief overview of the scope of work of this dissertation. We present an integrated approach which focuses on the development of various methods for modeling, analysis and verification of RTSS.

3.2 Problem definition

3.2.1 Modeling

Development of RTSS using UML/MARTE is gaining popularity day by day. MARTE model-driven approach can be used to describe timing constraints of RTSS. In many situation, concrete syntactical forms are necessary to represent complex data structure. It is not possible to model these types of complex data using standard UML. In this section, we model RTSS using tuple relational calculus and in Chapter 4 this is implemented by extending the MARTE metamodel using Value Specification Language (VSL).

Model Representation

We present a model for RTSS which is represented as a set

Model = {UCD, SEQD, TIMED, CLASSD}, where

UCD = {UCDi|1 $\leq$ i $\leq$ n} is finite set of Use Case diagrams

SEQD = {SeqDi | 1 $\leq$ i $\leq$ n} is finite set of Sequence diagrams for Use Case

TIMED = {TimeDi | 1 $\leq$ i $\leq$ n} is finite set of Timing diagrams for SEQD

CLASSD={ClassDi| 1 $\leq$ i $\leq$ n } is finite set of Class diagrams for TIMED

Representation of UCD

Use Case Diagram (UCD) is defined as a set

UCD = {A, UC, R, CO}, where

- A is a finite set of actors where A = {Ai | 1 $\leq$ i $\leq$ n},
- UC is a finite set of Use Cases where UC = {uCi | 1 $\leq$ i $\leq$ n},
- R is a finite set of relationships where R = {Assoc, Include, Extend, GenUC, GenAc},
- CO is a finite set of timing constraints for UCD where CO = {COi | 1 $\leq$ i $\leq$ n},
- COi is a finite set of timing constraints for a Use Case uCi where COi = {COij | 1 $\leq$ i $\leq$ n and 1 $\leq$ j $\leq$ n}

Representation of SEQD

$SeqD_{uCi}$ is defined as a finite set of Sequence diagrams corresponding to Use Case uCi.

$SeqD_{uCi}$ = {$SeqD_{uCi1}$, $SeqD_{uCi2}$,.........., $SeqD_{uCin}$ | uC ∈UC} where $SeqD_{uCi}$ ∈ SEQD

$SeqD_{uCin}$ is defined as a tuple

$SeqD_{uCin}$ = Ps, E, V, l, O, C, S where

- Ps is a set of lifelines denoting participants involved in an interaction where Ps = {PSi | 1 ≤ i ≤ n}
- E is a set of events where each event corresponds to sending or receiving a message where E = {Ei | 1 ≤ i ≤ n}
- V is a finite set of edges. V is defined as a link between two Ps. So V can be represented as {(e,e') | e,e' ∈ E and e'≠ e}. V = {Vi | 1 ≤ i ≤ n}
- l is a labeling function which assigns each v ∈ V a message name m with m = l (v)
- O is a function which maps each e ∈ E to the participant it belongs to
- C is a set of boolean of form t(e) − t(e') ≤ ≤ d which represents the timing constraints enforced on SeqDuCin
- S is a finite set of states to which participant goes where S = {Si | 1 ≤ i ≤ n}

There is an ordering relation over E in a participant. All events related to one participant are timely ordered. For any two distinct events ei and ej, let ei < ej denotes that ej occurs after ei if and only if (e,e') ∈ V.

We define NSeqD to be the set of all message names occurring in the Sequence diagram and denote NSeqD,p set of all message names sent or received by the participant p ∈ Ps of the Sequence diagram. We define NSeqDstate,p set of all states associated with the participant p $\in$ Ps of the Sequence diagram. We denote the event of receiving message mi as r(mi) and the event of sending message mi as s(mi).

Representation of TIMED

$Time_{Din}$ describes a Timing diagram corresponding to Sequence diagram $SeqD_{uCin}$.
$Time_{Din}$ is defined as a tuple,
$Time_{Din}$ = {Pt, M, D, S}, where

- Pt is a set of lifelines denoting participants involved in an interaction where Pt = {PTi | 1 $\leq$ i $\leq$ n}
- M is a set of messages and methods transferred between two pti where M = {Mi | 1 $\leq$ i $\leq$ n}
- D is the finite set of timing constraints where D = {Di | 1 $\leq$ i $\leq$ n}
- S is a finite set of states in the lifeline of Pt where S = {Si | 1 $\leq$ i $\leq$ n}

We defineN_{timeD} to be the set of all message names occurring in the Timing diagram and denote $N_{TimeD,p}$ set of all message names sent or received by the participant p $\in$ Pt of the Timing diagram. We define $N_{TimeDstate,p}$ as a set of all states associated with the participant p $\in$ Pt of the Timing diagram.

Representation of CLASSD

$Class_{Din}$ describes a Class diagram corresponding to Timing diagram $Time_{Din}$.
$Class_{Din}$ is defined as a tuple,
$Class_{Din}$ = {C, V, MD, T}, where

- C is the finite set of classes where C = {Ci | 1 $\leq$ i $\leq$ n}
- V is the finite set of variables where V = {Vi | 1 $\leq$ i $\leq$ n}
- MD is a set of methods where MD = {MDi | 1 $\leq$ i $\leq$ n}
- T is the finite set of timing constraints where T = {Ti | 1 $\leq$ i $\leq$ n}

3.2.2 Consistency

Real-time software development life cycle consists of different stages, which are correlated to each other and represent common aspects. The design model of RTSS consists of a set of diagrams which represent different aspects of the same software system and there are chances of inconsistencies within the models as they depict overlapping characteristics. A set of consistency criteria is introduced to ensure consistency within the design models. In this dissertation, we develop a framework to ensure inter diagram consistency such that the following rules are valid.

We are introducing inter diagram consistency rules based on the following tuple relations defined in section 3.2.1.

UCD = {A, UC, R, CO},

$SeqD_{uCin}$ = Ps, E, V, l, O, C, S

$Time_{Din}$ = {Pt, M, D, S},

$Class_{Din}$ = {C, V, MD, T},

Rule 1: UCD-$SeqD_{uCin}$: Ps $\subseteq$ A && C $\subseteq$ CO

Rule 2: $SeqD_{uCin}$- $Time_{Din}$: Pt $\subseteq$ Ps && D $\subseteq$ C

Rule 3: $Time_{Din}$-$Class_{Din}$: C $\subseteq$ Pt && MD $\subseteq$ M && T $\subseteq$ D

3.2.3 Verification

In the domain of requirements engineering, the term traceability is usually defined as the ability to follow the traces or, in short, to trace to and from requirements [98]. Formally, it can be defined as follows.

RT1 : RequiredTrace[origin $\rightarrow$ UCD;

destination $\rightarrow$ $SeqD_{uCin}$, $Time_{Din}$, $Class_{Din}$] where

RT1 is an element of the class RequiredTrace (required traceability), which requires

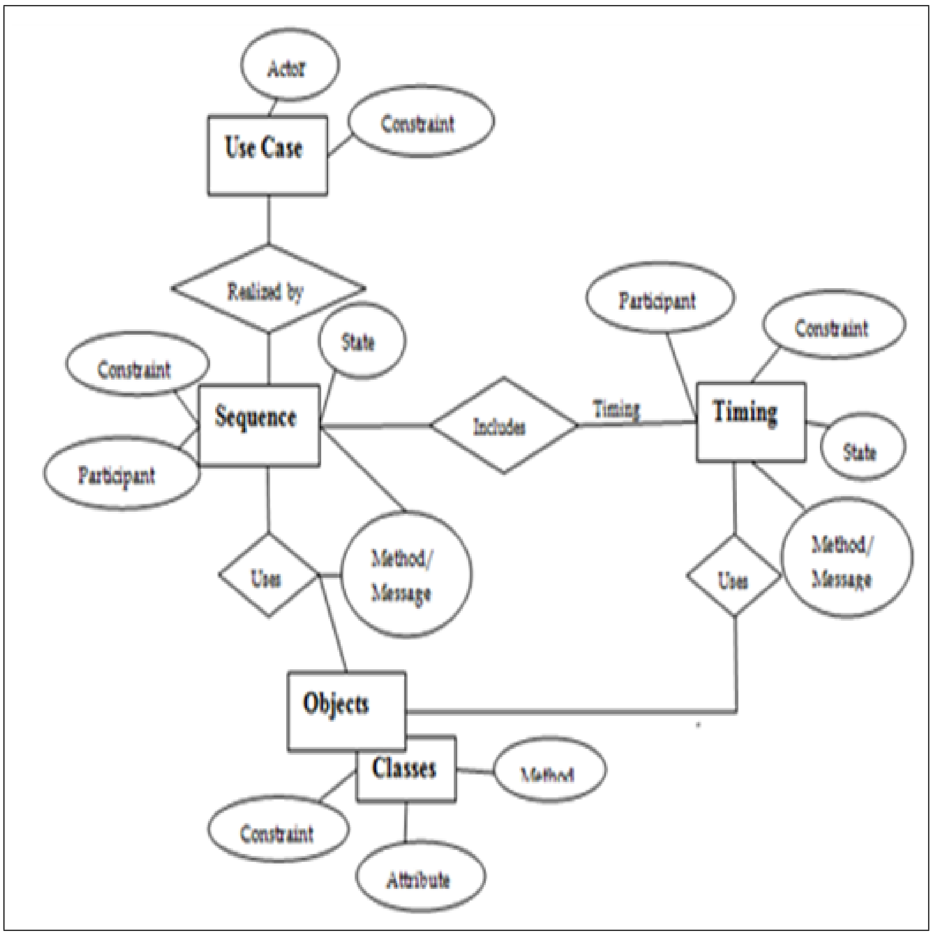


FIGURE 3.1: Relationship among UML models

traceability between UCD(Use case diagram) and $SeqD_{uCin}$, $Time_{Din}$, $Class_{Din}$ (Sequence, Timing and Class diagrams).

UML diagrams are used to model different aspects, however as they all together represent the same system, there may exist some overlapping areas that represent some common characteristics. Hence, the diagrams are related to each other and together they seamlessly integrate to represent the system as a whole. The UML diagrams are considered as entities and their relationships are modeled as an E-R diagram in Figure 3.1. Attributes of the entities are shown in Table 3.1.

This E-R diagram reveals the dependencies between the UML diagrams considered

Table 3.1: Entities and their attributes

Entities	**Attributes**
Use Case	Actor, Constraint
Sequence	Participant, State, Method/Message,Constraint
Timing	Participant, State, Method/Message, Constraint
Class	Attribute, Method, Constraint

here. The verification of requirement traceability is formulated based on the relationship discussed in this section.

In this section, requirement traceability is defined in formal way. We have developed a framework for verifying requirements with respect to design specifications. UML/MARTE is considered as the model for designing RTSS. The present framework enables developers to keep track of every requirement. Each requirement is traced in the design, which will help developers to identify unimplemented requirements on one hand and trace the path for implemented requirements. This will help in managing changes to requirements as well. We have developed metrics based requirements traceability for verification of RTSS to show how requirements related to timing constraints have been traced automatically from the use case diagram through the sequence diagram to the timing diagram and finally to the class diagram by introducing semantic actions in the grammar. Chapter 6 presents the details of this framework.

3.2.4 Schedulability analysis

A RTSS is schedulable if all tasks finish their execution before their respective deadlines. Analyzing the utilization of the processor for a given set of tasks can give an indication of schedulability. The utilization test of a system is performed by using the formula [70]:

$\sum_{i=1}^{n} = \frac{Ci}{Ti} \leq$ U(N) where

Ci is the worst-case execution time of task i

Ti is the period of task i

and U(N) = N($2^{1/n}$-1) is the utilization bound for N tasks

This dissertation discusses the ability of UML and its profile to determine the schedulability of RTS. This work puts stresses on the issues related to the deadlock of resource access control protocols. The Priority Inheritance Protocol is used for sharing critical resources but it does not prevent deadlock if nested critical sections are present. This shortcoming is represented using one UML model. Further, the Priority Ceiling, Stack Based Priority Ceiling and Stack Based Preemption Ceiling Protocols are used to overcome the difficulty using other improved models. Chapter 7 presents the details of this framework.

3.3 Conclusion

This chapter presents a broad overview of the scope of work presented in this dissertation. Several different new modeling, analysis and verification techniques presented in the subsequent chapters have been discussed briefly. The functional requirements of the system have been described. Schedulability analysis of various resource access control protocols are also highlighted.

Chapter 4

Modeling of Real-Time Software Systems

4.1 Introduction

Real-Time Systems interact with their environments using time constrained input/output signals. Functional misbehavior or a deviation from the specified time constraints may have catastrophic consequences. Developing a real-time embedded system is a sophisticated and complex task. It is crucial that for such systems, the software should be designed with a robust and sound architecture, which is capable of capturing not only the functional aspects but also the time constraints that are specific to a real-time software system. The increasing complexities of ubiquitous real-time systems require an adequate modeling language.

UML, a widely used visual object-oriented modeling language, has proven to be effective and suitable for real-time systems. It is now become one of the most widely used modeling languages for industrial software systems, essentially because it is a semiformal notation, relatively easy to use and well supported by tools.

It is seen that the constructs provided by the standard UML and the existing UML extension mechanisms are not enough in several applications such as medical domain, communication domain etc. i.e. this language is not adequate to model domain-specific constructs. In order to fill up this gap, OMG has introduced a mechanism called profiles which have the capabilities to extend the UML metamodel for different purposes. A profile is a generic extension of the base modeling language, it does not contradict the original language's semantics. Concretely, a UML profile is implemented as a set of stereotypes, tag values and constraints applied to the elements inside UML, e.g. classes, operations, activities, interactions.

New semantics have also been provided by this extension. UML/MARTE profile has been introduced by OMG to extend the UML standard by providing new concepts that are commonly encountered in real-time and embedded systems (RTES). The MARTE UML profile extends UML to add support for modeling elements and concepts that are specific to RTES.

This chapter focuses on the modeling of software development for real-time system using Use case, Sequence, Timing and Class diagrams by extending the UML/-MARTE metamodel.

4.2 Extension of UML/MARTE

UML is a multi-purpose semi-formal language with many notational constructs. UML provides extension mechanisms to allow the user to model real-time software systems if the current UML technique is not semantically sufficient to express the systems. These extension mechanisms are stereotypes, tagged values, and constraints.

4.2.1 Stereotype

Stereotypes allow the definition of extensions to the UML vocabulary, denoted by <<stereotype-name>>. The base class of a stereotype can be different model elements, such as class, attribute, and operation. A stereotype groups tagged values and constraints under a meaningful name. When a stereotype is branded to a model element, the semantic meaning of the tagged values and the constraints associated with the stereotype are attached to that model element implicitly. A number of possible uses of stereotypes have been classified in several research works.

4.2.2 Tagged value

Tagged values extend model elements with new kinds of properties. Tagged values may be attached to a stereotype, and this association will navigate to the model element to which the stereotype is branded. The format of a tagged value is a pair of tag name and an associated value, i.e. {name: value}. The tagged values attached to a stereotype must be compatible with the constraints of the stereotype's base class.

4.2.3 Constraint

Constraints add new semantic restrictions to a model element. Typically constraints are written in the Object Constraint Language (OCL). Constraints attached to a stereotype imply that all model elements branded by that stereotype must obey the semantic restrictions which constraints state. Note that the constraints attached to a stereotyped model element must be compatible with the constraints of the stereotype and the base class of the model element. A profile is a stereotyped package that contains model elements that have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged values, and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends.

4.2.4 VSL

VSL is coupled with the Non-Functional Properties (NFP) sub-profile of MARTE, which can be used to specify the complex types behind non functional properties of a system [90]. VSL covers all the expressibility needs implied by a typical real-time design flow. The usage of VSL is quite straightforward for the specification of values. In the case of constraint specifications, using VSL may appear less intuitive, especially if the constraint concerns timing aspects. VSL is used most often in the following situations:

- Assigning a value to a class or stereotype attribute
- Writing constraints, typically ones that involve values of object and class attributes
- Defining complex data types

4.3 Case Study

Patient Ventilation System (PVS) is considered as a running example of RTSS. This section gives an overview of the PVS and the functional specifications of the system are shown. PVS helps an anesthetized and paralyzed patient to breathe properly by controlling the delivery of fresh mixed gases to the patient and the removal of waste gases, as illustrated in Figure 4.1 [31].

Breathing circuit on the inspiratory side accepts the fresh mixed gas (containing between 21% and 100% O_2) which mixes with the “scrubbed” gas returning within

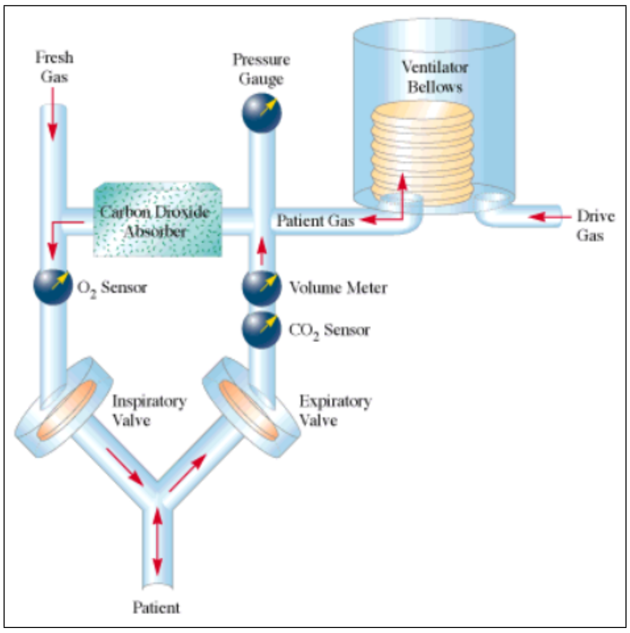


FIGURE 4.1: Block diagram of PVS

the circuit to the patient. This gas enters the expiratory limb of the breathing circuit so that the patient can ultimately exhale this gas. The movement of gas is powered by the bellows assembly (of the patient ventilator) which pushes the bellows down to force inhalation and reduces the pressure in the drive gas. This process allows a passive exhalation and a refilling of the bellows.

Specifying the bound between two ventilation cycles, this timing constraint has a soft constraint and a hard constraint. The soft constraint is fourteen ventilations per minute, and the hard constraint is twelve ventilations per minute. In this closed breathing circuit, the expired gases are pushed through a soda lime canister that removes CO_2 from the mixture. This scrubbed gas is then mixed with some amount of new, fresh gas replacing the removed oxygen. In addition to the movement of gases, ventilators usually perform some machine monitoring to enhance patient safety.

The common parameters monitored are:

- O_2 concentration in the inspired limb of the breathing circuit (fiO_2)
- CO_2 concentration in the expired limb of the breathing circuit (etCO_2)

- Volume flow through the patient
- Breathing circuit pressure sensor

The ventilator operates in three modes:

- Ventilation mode (normal operating mode for the machine)
- Configuration mode (user can configure the machine, set alarm limits, and so on)
- Service mode (service personnel can update software, calibrate or replace the sensors, and so forth)

Considering the following basic functions, the requirements have been identified (Table 4.1), which will be considered further in the design phase.

- Delivering ventilation to the patient with value 12/min
- Checking alarm on the critical event occurred on patient
- Monitoring breathing circuit
- Displaying informational alarms for a period no more than two minutes
- Displaying warning alarms until acknowledged by the user within 30 sec
- Re-announcing critical alarms after being dismissed out if the originating condition still exists
- Delivering a specified volume per breath (known as Tidal Volume)
- Setting a specified Inspiration-to-expiration time ratio (known as I:E Ratio)
- Setting a specified time for inspiration (Inspiration Time)
- Setting a pause between breaths (Inspiratory Pause)
- Setting a rate of respiration (Respiration Rate)
- Setting max pressure limit
- Setting inspiratory flow

The requirements document is formed based on the above functional requirements with each requirement being assigned a unique identifier for enabling traceability into design. The requirements document is shown in Table 4.1.

TABLE 4.1: Requirements document

RID	Description
R01	Physician can deliver ventilation to patient with value 12/min
R02	Physician can check alarm on critical event occurred on patient
R03	Informational alarm is displayed for a period no more than two minutes
R04	Caution alarm is displayed until acknowledged by the user within 30 sec.
R05	Critical alarm is re-announced after being dismissed out if the originating condition still exists.
R06	Physician can set various ventilation parameters
R07	Physician can monitor breathing circuit
R08	Physician can set inspiratory flow
R09	Physician can deliver a specified volume per breath
R10	Physician can set a specified inspiration-to-expiration time ratio
R11	Physician can set a specified time for inspiration
R12	Physician can set a pause between breaths
R13	Physician can set a rate of respiration
R14	Physician can set max pressure limit

4.4 Modeling with extended UML/MARTE

This chapter presents modeling of a real-time software system using the UML/-MARTE based Use case, Sequence, Timing and Class diagrams; discusses the representations, and shows how they can be enhanced by adding new elements to the underlying design notation. First, the modeling of new user defined data types with VSL is discussed and then representation of different UML diagrams by using those new data types by extending the MARTE metamodel are also discussed. We have used "Altova" modeling tool to draw the timing diagram and rest of the diagrams are drawn using "MagicDraw" modeling tool.

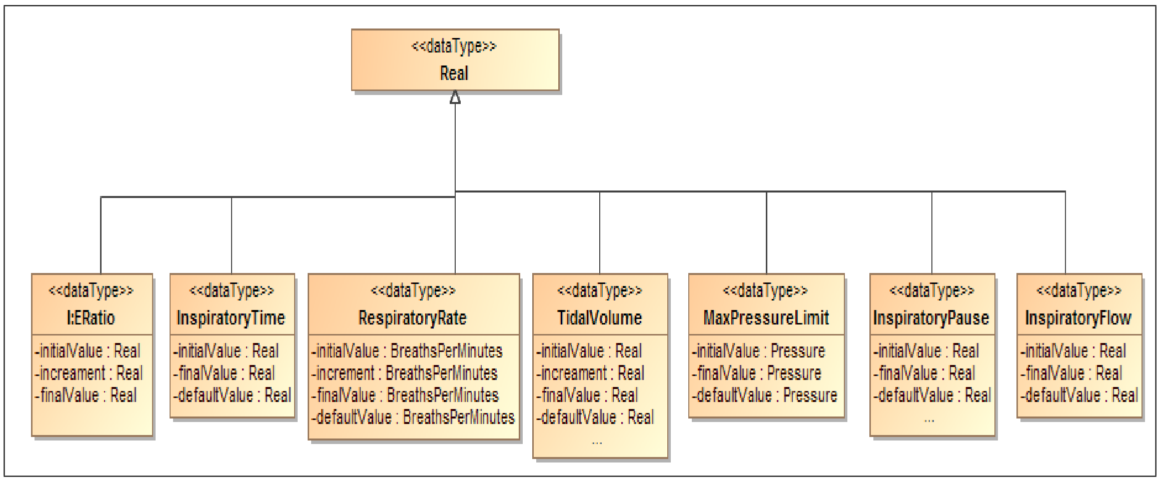


FIGURE 4.2: Defining new user defined structured data types by extending MARTE

4.4.1 Structured data types with VSL

Sometimes, concrete syntactical forms are required to represent complex data structure. Using standard UML, we cannot model these types of complex data. VSL expression is very useful to resolve this difficulty. VSL can be used to extend the UML primitive data types.

In our model, we have defined seven new structured data types; I:ERatio, InspiratoryTime, RespiratoryRate, TidalVolume, InspiratoryFlow, InspiratoryPause and MaxPressureLimit as subclasses of the MARTE Real primitive type.

InspiratoryTime data type has three attributes, initialValue, finalValue and defaultValue (Figure 4.2). As all of these attributes are expressed using real numbers, we make InspiratoryTime a subclass of the MARTE Real primitive type. Likewise, other data types are also defined and self-explanatory.

4.4.2 Adding new physical data types

The standard MARTE library does not contain some of the desired physical data types required for the patient ventilation system. For this purpose, it is required to

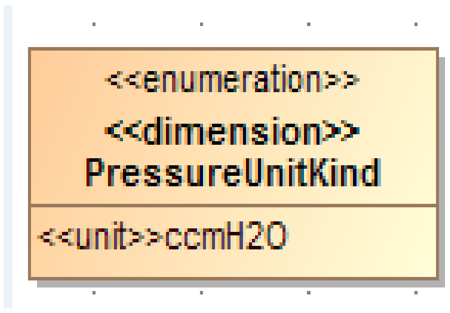


FIGURE 4.3: PressureUnitKind dimension definition

define new custom types [90].

This involves the following number of steps-

1. Defining the appropriate physical units and their concrete representation (e.g., seconds, grams, etc.)

2. Defining the unit type (dimension) of physical units required (e.g., energy, length, volume etc.)

3. Defining the desired physical type, that is, the combination of value and unit that represents the desired physical quantity

Step 1: Defining the physical units and their concrete representation

It is required to define the units we are interested in including in our models. Let us assume that this example considers only two types of units which are designated by the literals “ccmH2O” and ”bpm”.

Step 2: Defining the unit type (Dimension) of the physical type

Specifying the unit type, or dimension will consist of a UML enumerated list annotated with the stereotype <<dimension>> and its literal will be annotated with the stereotype <<unit>> In our example, we have defined a new PressureUnitKind (Figure 4.3) and BpmUnitKind (Figure 4.4) enumerations, which are stereotyped with the predefined MARTE Dimension to differentiate it from other types of enumerations.

Step 3: Defining the new physical type

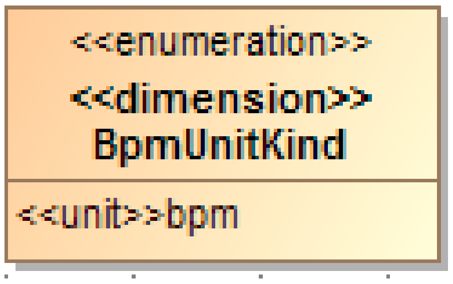


FIGURE 4.4: BpmUnitKind dimension definition

The following steps are used to define a new physical data type:

(i) Selecting the appropriate physical value type, based on the kind of value to be specified
(ii) Creating a new subclass of that type and adding the corresponding unit attribute
The stereotype <<nfpType>> is used for declaring types of non-functional property. This is an extension of the concept of UML <<dataType>>.
The MARTE Nfp_Type stereotype defines the general format of NFP types. It helps to differentiate between existing data types in MARTE library and other kinds of custom data types.
This stereotype supports three kinds of attributes:
- valueAttrib — This attribute contains the value of data type. This is usually a numerical quantity (integer or real), although, in some cases, it can be a string.
- unitAttrib — This attribute contains the physical measurement unit corresponding to the values. This is always an enumeration literal tagged with the Unit stereotype.
- exprAttrib — This optional attribute points to the attribute that contains a VSL expression to be used for various purposes.
This work considers only valueAttrib and unitAttrib.

In this chapter, we have chosen the existing MARTE NFP_Integer physical value type as our base class, because we want to be able to express integer pressure values (following step (i)). Further, we have derived a new class Pressure from the existing MARTE NFP_Integer physical value type (following step (ii)) as shown in Figure

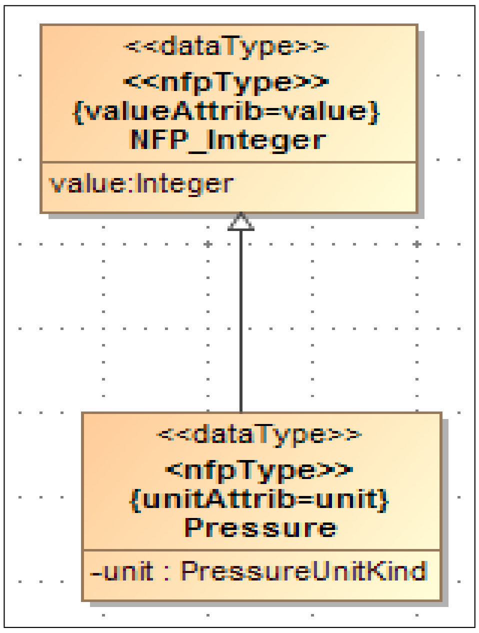


FIGURE 4.5: Defining the new Pressure physical type

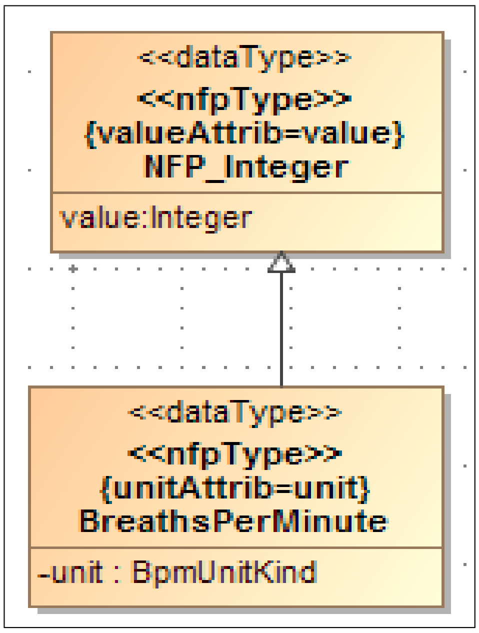


FIGURE 4.6: Defining the new BreathsPerMinutes physical type

4.5. Here, we have defined the new physical type, Pressure and unit will be ccmH2O. Similarly, we have also defined another new physical type, BreathsPerMinutes, unit of which is bpm as shown in Figure 4.6.

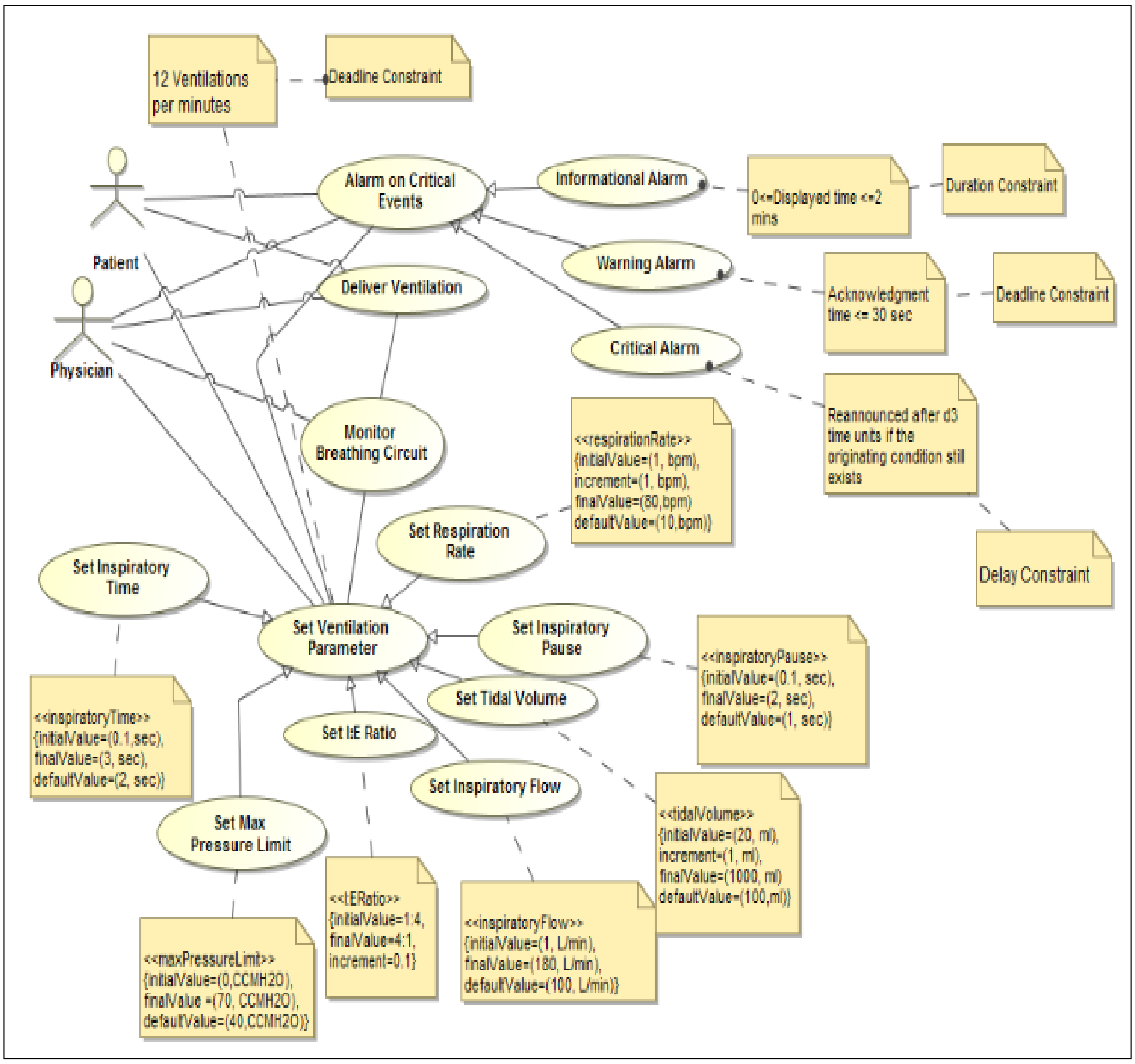

12 Ventilations per minutes
Deadline Constraint
Patient
Physician
Alarm on Critical Events
Informational Alarm
0<=Displayed time <=2 mins
Duration Constraint
Warning Alarm
Acknowledgment time <= 30 sec
Deadline Constraint
Deliver Ventilation
Critical Alarm
Reannounced after d3 time units if the originating condition still exists
Delay Constraint
Monitor Breathing Circuit
<<respirationRate>> {initialValue=(1, bpm), increment=(1, bpm), finalValue=(80,bpm) defaultValue=(10,bpm)}
Set Respiration Rate
Set Inspiratory Time
Set Ventilation Parameter
Set Inspiratory Pause
<<inspiratoryPause>> {initialValue=(0.1, sec), finalValue=(2, sec), defaultValue=(1, sec)}
Set Tidal Volume
<<inspiratoryTime>> {initialValue=(0.1,sec), finalValue=(3, sec), defaultValue=(2, sec)}
Set I:E Ratio
Set Inspiratory Flow
<<tidalVolume>> {initialValue=(20, ml), increment=(1, ml), finalValue=(1000, ml) defaultValue=(100,ml)}
Set Max Pressure Limit
<<IERatio>> {initialValue=1:4, finalValue=4:1, increment=0.1}
<<inspiratoryFlow>> {initialValue=(1, L/min), finalValue=(180, L/min), defaultValue=(100, L/min)}
<<maxPressureLimit>> {initialValue=(0,CCMH2O), finalValue =(70, CCMH2O), defaultValue=(40,CCMH2O)}

FIGURE 4.7: Use case diagram for Patient Ventilation System

4.5 Modeling of PVS with MARTE annotations

This section models the real-time patient ventilation system using Use case, Sequence, Timing and Class diagrams by extending MARTE metamodel to introduce new user defined data types.

These diagrams depict different views of the system. The requirements are captured in the Use case diagram which forms the beginning of the analysis phase. The Class diagram represents the structural aspects while Sequence and Timing diagrams correspond to the dynamic aspects of the system under consideration. Together they capture both the structural and behavioral characteristics of the system.

4.5.1 Use Case diagram for PVS

• Patient and Physician are considered as the main users who are modeled as actors in UML notation.

• The Use Case diagram (Figure 4.7) models the fourteen requirements (Table 4.1) as use cases. Fourteen use cases are (Deliver Ventilation, Alarm on Critical event, Informational Alarm, Warning Alarm, Critical Alarm, Monitor Breathing Circuits, Set Ventilation Parameter, Set Inspiratory Time, Set Tidal Volume, Set I:E ratio, Set Respiration rate, Set maximum pressure limit, Set Inspiratory pause, Set Inspiratory Flow) present in the Use case diagram.

• Two types of relationships are present among the use cases. They are Association and Generalization.

• Annotations are represented by using the existing physical data types present in the MARTE library as well as new user defined data types such as Pressure and BreathsPerMinutes by extending the MARTE metamodel. Those custom data types are used to represent the type attributes of various elements in our model, which have either pressure or BreathsPerMinutes characteristic.

• Following seven stereotypes exist in the Use case diagram
i) <<inspiratoryPause>>
ii) <<inspiratoryFlow>>
iii) <<inspiratoryTime>>
iv) <<tidalVolume>>
v) <<maxPressureLimit>>
vi) <<I:ERatio>>
vii) <<respirationRate>>

• The use case ”Set Inspiratory Flow” is annotated by specifying the stereotype <<inspiratoryFlow>>, Tags are initialValue=1 L/min, finalValue=180 L/min and defaultValue=100 L/min.

• Following different types of timing constraints are present in the Use case diagram.
i) 12 ventilations per minutes (deadline constraint)...............(I)
ii) Informational Alarm is displayed for a period no more than 2 minutes (duration constraint).
iii) Warning Alarm is acknowledged by the user within 30 secs (deadline constraint).
iv) Critical Alarm is reproduced after every 2 minutes if the originating condition still exists (delay constraint).

4.5.2 Sequence diagram for use case Set Ventilation Parameter

• The sequence diagram shows the dynamic behavioral view of the system where the messages are passed to and from the external world. The messages are ordered as per the sequence in which they occur.

• Similar to the use case diagram in Figure 4.7, the sequence diagram (Figure 4.8) is represented by using the existing physical data types present in the MARTE library and as well as new user defined data types such as Pressure and BreathsPerMinutes.

• Eight participants are (Timer, Inspiratory Valve, Compressor, Gas Mixer, Patient, Exhalation Valve, Parameter_Controller, Physician) present in the Sequence diagram.

• Some messages and methods are Set_Timer(), Set_Tidal_Volume(), Set_Inspiratory_Time(), Set_Max_Pressure_Limit(), Set_I:E_ratio(), Set_Respiration_Rate(), Set_Inspiratory_Pause(), open_valve(), close_valve() and Set_Inspiratory_Flow() etc.

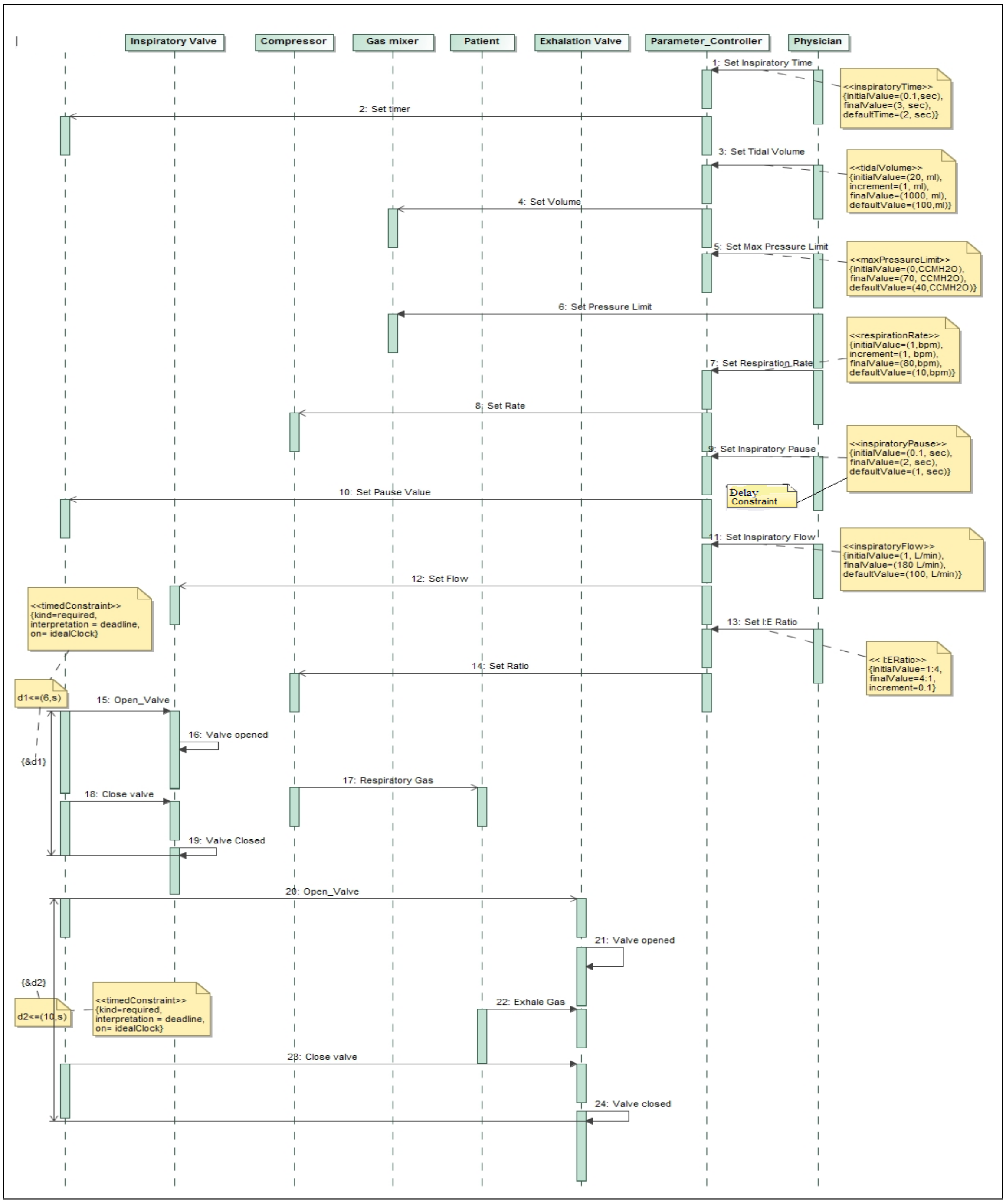

Inspiratory Valve
Compressor
Gas mixer
Patient
Exhalation Valve
Parameter_Controller
Physician
1: Set Inspiratory Time
<<inspiratoryTime>> {initialValue=(0.1,sec), finalValue=(3, sec), defaultTime=(2, sec)}
2: Set timer
3: Set Tidal Volume
<<tidalVolume>> {initialValue=(20, ml), increment=(1, ml), finalValue=(1000, ml), defaultValue=(100,ml)}
4: Set Volume
5: Set Max Pressure Limit
<<maxPressureLimit>> {initialValue=(0,CCMH2O), finalValue=(70, CCMH2O), defaultValue=(40,CCMH2O)}
6: Set Pressure Limit
<<respirationRate>> {initialValue=(1,bpm), increment=(1, bpm), finalValue=(80,bpm), defaultValue=(10,bpm)}
7: Set Respiration Rate
8: Set Rate
9: Set Inspiratory Pause
<<inspiratoryPause>> {initialValue=(0.1, sec), finalValue=(2, sec), defaultValue=(1, sec)}
Delay Constraint
10: Set Pause Value
11: Set Inspiratory Flow
<<inspiratoryFlow>> {initialValue=(1, L/min), finalValue=(180 L/min), defaultValue=(100, L/min)}
12: Set Flow
13: Set I:E Ratio
<< I:ERatio>> {initialValue=1:4, finalValue=4:1, increment=0.1}
14: Set Ratio
<<timedConstraint>> {kind=required, interpretation = deadline, on= idealClock}
d1<=(6,s)
15: Open_Valve
16: Valve opened
{&d1}
17: Respiratory Gas
18: Close valve
19: Valve Closed
20: Open_Valve
21: Valve opened
{&d2}
d2<=(10,s)
<<timedConstraint>> {kind=required, interpretation = deadline, on= idealClock}
22: Exhale Gas
23: Close valve
24: Valve closed

FIGURE 4.8: Sequence diagram for use case Set Ventilation Parameter

• Seven stereotypes exist in the Sequence diagram such as <<inspiratoryPause>>, <<inspiratoryFlow>>, <<inspiratoryTime>>, <<tidalVolume>> etc.

• The method "Set_Inspiratory_Flow" is annotated by the stereotype <<inspiratoryFlow>>, Tags are {initialValue = 1 L/min}, {finalValue =180 L/min} and {defaultValue = 100 L/min}.

Using the sequence diagram, we can represent timing constraints in a more informative way. PVS consists of timing constraints 12 ventilations per minutes, which is further sub divided into two timing constraints as follows.
o Duration between open and closed Inspiratory valve must be d1 units (duration constraint).
o Duration between open and closed Exhalation valve must be d2 units (duration constraint).
Hence, one ventilation must be completed within (d1+d2) units (where (d1+d2) $\leq$ 5s).(deadline constraint).
The timing constraints are represented by using the UML/MARTE [90] DurationObservation concept. In this case, the duration observation &d has an associated Duration element named d. A stereotyped constraint "TimedConstraint" is then applied to the duration element d.

In addition to deadline and duration constraints, the sequence diagram also consists of the following timing constraint.

o Delay between two breaths must be d units (delay constraint).

4.5.3 Timing diagram for use case Set Ventilation Parameter

The timing diagram is a simple representation with time along with the horizontal axis and object state or attribute value along the vertical axis. The absolute timing

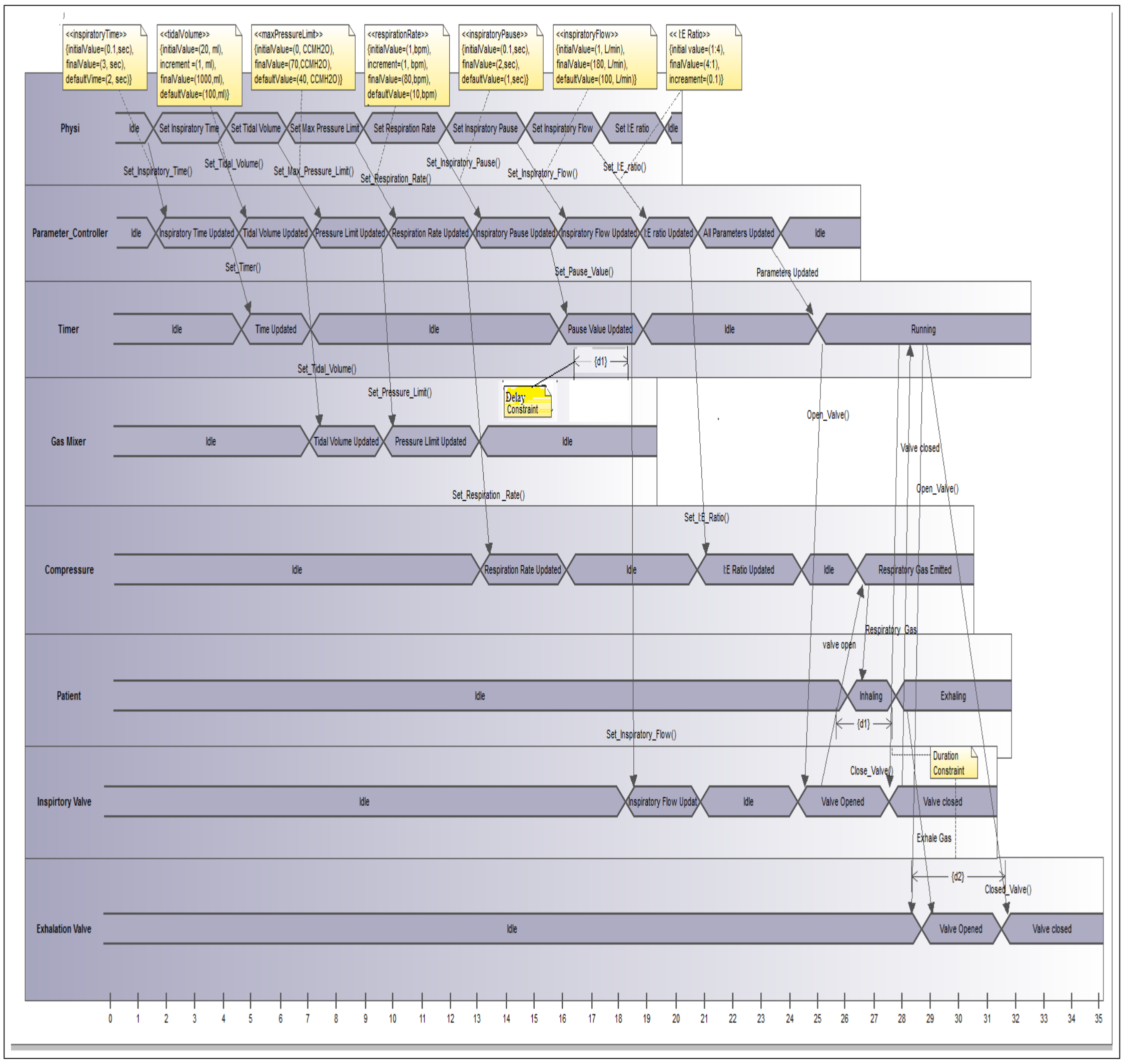


FIGURE 4.9: Timing diagram for use case Set Ventilation Parameter

of events, state changes and the relative timing among the lifelines are clearer and readable in timing diagrams. The relative timing between two messages can be specified appropriately.

• The timing diagram (corresponding to sequence diagram (Figure 4.8)) can be represented by using MARTE annotations (Figure 4.9).

• The time scale for this Timing diagram is represented from 0 to 35.

• Eight participants (Physician, Parameter_Controller, Timer, Gas_Mixer, Compressor, Patient, Inspiratory Valve, Exhalation Valve) are present in the Timing diagram.

• Some methods are Set_Timer(), Set_Tidal_Volume(), Set_Inspiratory_Time(), Set_Max_Pressure_Limit(), Set_I:E_ratio(), Set_Respiration_Rate(), Set_Inspiratory_Pause(), open_valve(), close_valve() and Set_Inspiratory_Flow() etc.

• The method "Set_Tidal_Volume ()" is annotated by the stereotype <<tidalVolume>> with tag values {initialValue = 20 ml}, {finalValue = 1000 ml}, {defaultValue = 100 ml} and {incerement = 1 ml}.

• The following different timing constraints are present in the Timing diagram.

1. Duration between open and closed Inspiratory valve must be d1 units (duration constraint).
2. Duration between open and closed Exhalation valve must be d2 units (duration constraint).
3. One ventilation must be completed within (d1+d2) units (deadline constraint).
4. Delay between two breaths must be d units (delay constraint).

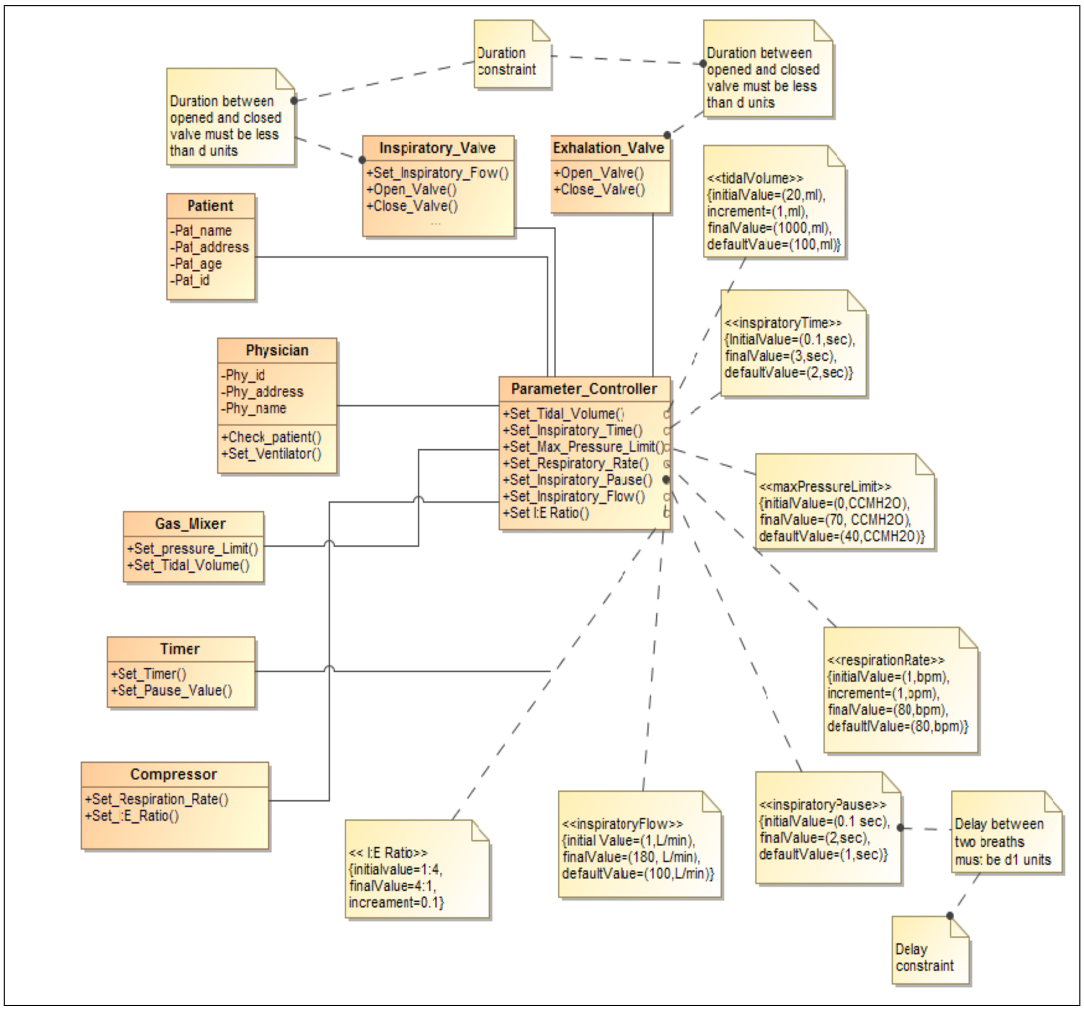

Duration constraint
Duration between opened and closed valve must be less than d units
Duration between opened and closed valve must be less than d units
Inspiratory_Valve
+Set_Inspiratory_Fow()
+Open_Valve()
+Close_Valve()
...
Exhalation_Valve
+Open_Valve()
+Close_Valve()
<<tidalVolume>> {initialValue=(20,ml), increment=(1,ml), finalValue=(1000,ml), defaultVaue=(100,ml)}
Patient
-Pat_name
-Pat_address
-Pat_age
-Pat_id
<<inspiratoryTime>> {InitialValue=(0.1,sec), finalValue=(3,sec), defaultValue=(2,sec)}
Physician
-Phy_id
-Phy_address
-Phy_name
+Check_patient()
+Set_Ventilator()
Parameter_Controller
+Set_Tidal_Volume()
+Set_Inspiratory_Time()
+Set_Max_Pressure_Limit()
+Set_Respiratory_Rate()
+Set_Inspiratory_Pause()
+Set_Inspiratory_Flow()
+Set I:E Ratio()
<<maxPressureLimit>> {initialValue=(0,CCMH2O), finalValue=(70, CCMH2O), defaultValue=(40,CCMH2O)}
Gas_Mixer
+Set_pressure_Limit()
+Set_Tidal_Volume()
Timer
+Set_Timer()
+Set_Pause_Value()
<<respirationRate>> {initialValue=(1,bpm), increment=(1,bpm), finalValue=(80,bpm), defaultValue=(80,bpm)}
Compressor
+Set_Respiration_Rate()
+Set_I:E_Ratio()
<<inspiratoryFlow>> {initial Value=(1,L/min), finalValue=(180, L/min), defaultValue=(100,L/min)}
<<inspiratoryPause>> {initialValue=(0.1 sec), finalValue=(2,sec), defaultValue=(1,sec)}
Delay between two breaths must be d1 units
<< I:E Ratio>> {initialvalue=1:4, finalValue=4:1, increament=0.1}
Delay constraint

FIGURE 4.10: Class diagram for use case Set Ventilation Parameter

4.5.4 Class diagram for use case Set Ventilation Parameter

Class diagram models static structural aspects of the system.

- Figure 4.10 represents the class diagram (corresponding to the timing diagram (Figure 4.9)) by extending the MARTE metamodel to incorporate newly user defined data types such as Pressure and BreathsPerMinutes.

- The classes are Physician, Parameter_Controller, Timer, Gas_Mixer, Compressor, Patient, Inspiratory Valve, Exhalation Valve.

- Some methods are Open_Valve(), Close_Valve(), Set_Pressure_Limit() etc.

• Some variables are Phy_id, Phy_name, Pat_id, Pat_name etc.

• The method "Set_Tidal_Volume()" is annotated by the stereotype <<tidalVolume>> with tag values {initialValue = 20 ml}, {finalValue = 1000 ml}, {defaultValue = 100 ml} and {incerement = 1 ml}.

• The following different timing constraints are present in the Class diagram.

1. Duration between open and closed Inspiratory valve must be d1 units (duration constraint).
2. Duration between open and closed Exhalation valve must be d2 units (duration constraint).
3. One ventilation must be completed within (d1+d2) units (deadline constraint).
4. Delay between two breaths must be d units (delay constraint).

4.6 Comparison with related work

In [31], Douglass has used the patient ventilation system as a case study to demonstrate how well UML can be used to represent a real-time system development problem. But, the work is limited to several important system specifications, parameters, and time-based constraints for modeling RTSS. In [116], the authors introduce a set of constructs for the designing of software architectures in the domain of real-time software systems by extending UML. In comparison to that, our work suggests additional constructs for timing constraints and develops a formal specification for automated analysis and verification. We have extended the MARTE metamodel to introduce new user defined data types required to model a real-time software system.

4.7 Threats to validity of the proposed approach

We have developed a comprehensive framework to introduce new user defined data types to model domain-specific constructs such as patient ventilation system. VSL is used to model new user defined data types by extending UML/MARTE. At present, only UML/MARTE and SysML profile support VSL. Also, the extension has been done for a specific domain. This framework may need to be modified for modeling different domain-specific constructs.

4.8 Conclusion

With the rapid growth in the size and complexity of real-time software systems and to guarantee the development of more reliable systems, modeling plays an important role in the software development life cycle. A minor fault in a model may lead to a major failure of real-time lifesaving systems. UML/MARTE which is a very effective and suitable profile for modeling design specifications of RTSS is gaining popularity day by day. In this chapter, we have modeled real-time patient ventilation system using UML/MARTE annotated Use case, Sequence, Timing and Class diagrams. This framework mainly focuses on the requirements analysis and design phase of a software development life cycle.

Chapter 5

Consistency in Real-Time Software Systems

5.1 Introduction

Software life cycle model consists of different interrelated stages, which contain some common attributes. Due to this, there is a requirement to verify inconsistencies among these stages. For a large and complex software system, where a specification consists of different parts and each part focuses on a specific view of the system, consistency checking is a prerequisite for the correct execution of the system as a whole. By checking consistency, we ensure that the properties of the different stages of a specification are consistent with each other and they do not oppose. The same holds for RTSS as well. In order to make our model more robust, we introduce a set of consistency rules within the design models.

UML is a visual modeling language where the different diagrams represent different but partly overlapping views of the system. Hence it is necessary to ensure that the overlapping diagrams represent the same common characteristics. Otherwise, the design becomes inconsistent. We have defined a set of consistency rules for some of the commonly used diagrams to achieve this objective. Violation of any rule renders the design to be inconsistent. This has been achieved by proposing formal approach to capture the consistency among different UML diagrams of the model developed in the previous section. Formal specification of a model enhance the use of automated tools that ease the development process from analysis through coding. This is additionally valid for RTSS, whose behavioral viewpoints can frequently be portrayed through UML.

In order to make our model (developed in Chapter 4) more robust, we propose a set of consistency rules within the design model handling common attributes. Further, we present formal approach for the automated verification of the consistency rules and also develop simulator tool to provide the decision of consistency or inconsistency. This chapter presents the details of the automated tools. The main functional block, code snippets and snapshots of some of the screens have been presented here.

5.2 Consistency among UML Diagrams

UML diagrams are composed of a finite set of elements and related through the existence of common elements. Therefore, there also exist a finite number of relationships, based on which, consistency rules are formulated. Since the order of relationship is irrelevant, both the relationships SQ-CL and CL-SQ represent the same relation between class and sequence diagrams. The total three relations are considered in this dissertation. However, for each relationship there are a number of common elements. The set of 3 relationships and common elements are tabulated in Table 5.1.

Thus from that table, it becomes evident that a total of 3 unique binary relationships exist between the four diagrams based on the existence of one or more common element(s) representing some common aspect of the system. For each of these, there exists a relation for each common element, which forms the basis of definition of the

TABLE 5.1: Consistency among UML diagrams

Sl. No.	Related Diagrams	Common Elements	Description
1	UC – SQ	u, a, st, tc	Use case diagram and Sequence diagram are related to each other based on use case, actor, stereotype and timing constraint
2	SQ -TM	c, m, st, tc	Sequence diagram and Timing diagram are related to each other based on class, method, stereotype and timing constraint
3	TM -CL	c, m, st, tc	Timing diagram and Class diagram are related to each other based on class, method, stereotype and timing constraint

consistency rules among the diagrams. Here we develop the rules for every individual common element and getting total 14 rules.

5.2.1 Consistency Rules among UML diagrams

The following consistency rules between different UML diagrams have been developed to ensure consistency in the design.

5.2.1.1 Use case diagram and Sequence diagram consistency

The following rules have been developed for ensuring consistency between Use case diagram and Sequence diagram.

Rule 1: Each Sequence diagram realizes a use case which should be present in the Use case diagram.

$\forall u | u \in SQ \Rightarrow u \in U$

Rule 2: Actors associated with a use case should occur in the corresponding Sequence diagram.

$\forall a | a \in AU \Rightarrow a \in AS$

Rule 3: All stereotypes associated with a Use Case must be present in the corresponding Sequence diagram.

$\forall st | st \in SU \Rightarrow st \in SS$

Rule 4: Different types of timing constraints (such as deadline, duration etc.) associated with a Use Case must be present in the corresponding Sequence diagram.

$\forall tc | tc \in CU \Rightarrow tc \in CS$

5.2.1.2 Sequence diagram and Timing diagram consistency

The following rules have been developed for ensuring consistency between Sequence diagram and Timing diagram.

Rule 5: Any Timing diagram can be realized from only one Sequence diagram.

$\forall sq | sq \in TM \Rightarrow sq \in SQ$

Rule 6: All lifelines in Sequence diagram must be present as participants in the corresponding Timing diagram.

$\forall p | p \in PS \Rightarrow p \in PT$

Rule 7: All messages associated with Sequence diagram should occur in the corresponding Timing diagram.

$\forall m | m \in MSS \Rightarrow m \in MST$

Rule 8: All methods associated with Sequence diagram must be present in the corresponding Timing diagram.

$\forall m1 | m1 \in MS \Rightarrow m1 \in MT$

Rule 9: All stereotypes associated with a Sequence diagram must be present in the corresponding Timing diagram.

$\forall st | st \in SS \Rightarrow st \in ST$

Rule 10: Different types of timing constraints associated with a Sequence diagram must be present in the corresponding Timing diagram.

$\forall tc | tc \in CS \Rightarrow tc \in CT$

5.2.1.3 Timing diagram and Class diagram consistency

The following rules have been developed for ensuring consistency between Timing diagram and Class diagram.

Rule 11:

All participants associated with Timing diagram must be defined as classes in the corresponding Class diagram.

$\forall p | p \in PT \Rightarrow p \in C$

Rule 12:

All methods associated with Timing diagram should occur in the corresponding Class diagram.

$\forall m | m \in MT \Rightarrow m \in MC$

Rule 13:

All stereotypes appearing in Timing diagram must be present in the corresponding Class diagram.

$\forall st | st \in ST \Rightarrow st \in SC$

Rule 14: Different types of timing constraints associated with the Timing diagram must be present in the corresponding Class diagram.

$\forall$tc|tc$\in$CT$\Rightarrow$tc$\in$CC

5.3 Formal specification for the automated checking of inter-diagram consistency

In this dissertation, we are considering formal grammar (using CFG representation) of four of the most commonly used UML diagrams - Use case, Sequence, Timing and Class diagrams for the automation of consistency checking. Semantic actions in the formal approach automatically find out inconsistencies among those UML diagrams at the early stage of the design. This formal specification is verified by ANTLR of version 3.1.3. For this purpose antlr.jar version 3.4 has been used which supports ANTLRWorks version 1.2.1 editor. This jar includes all the required files to verify if a grammar written in ANTLR specific language compiles successfully. To verify the grammar, first it has been edited in the ANTLRWorks editor. Then it is debugged using the debug button to compile it.

There are obvious reasons for using formal portrayal. Formal strategies are expected to systematize and bring thoroughness at every phase of software development. This helps us to abstain from disregarding basic issues, gives a standard intends to record different suppositions and choices, and structures a reason for consistency among many related exercises. By giving exact and unambiguous portrayal systems, formal strategies encourage the understanding required to blend the different periods of programming improvement into an effective attempt. Formal verification avoids the blemishes that crawled into their informal reasoning. We accomplished this objective.

Further, formal notations of a model enhance the use of automated tools that ease the development process from analysis through coding. This is additionally valid for RTSS, whose behavioral viewpoints can frequently be portrayed through UML. Consequently, it is fascinating to consider how well UML is adjusted to the real-time context.

5.3.1 Introduction to ANTLR

ANTLR [95] is a parser generator in computer-based language recognition. Alike LEX/YACC duo, ANTLR is a "translator generator tool". Language grammars for ANTLR are defined either in ANTLR syntax or in a special Abstract Syntax Tree (AST) syntax. The ANTLR syntax is more like YACC and Extended Backus-Naur Form (EBNF).

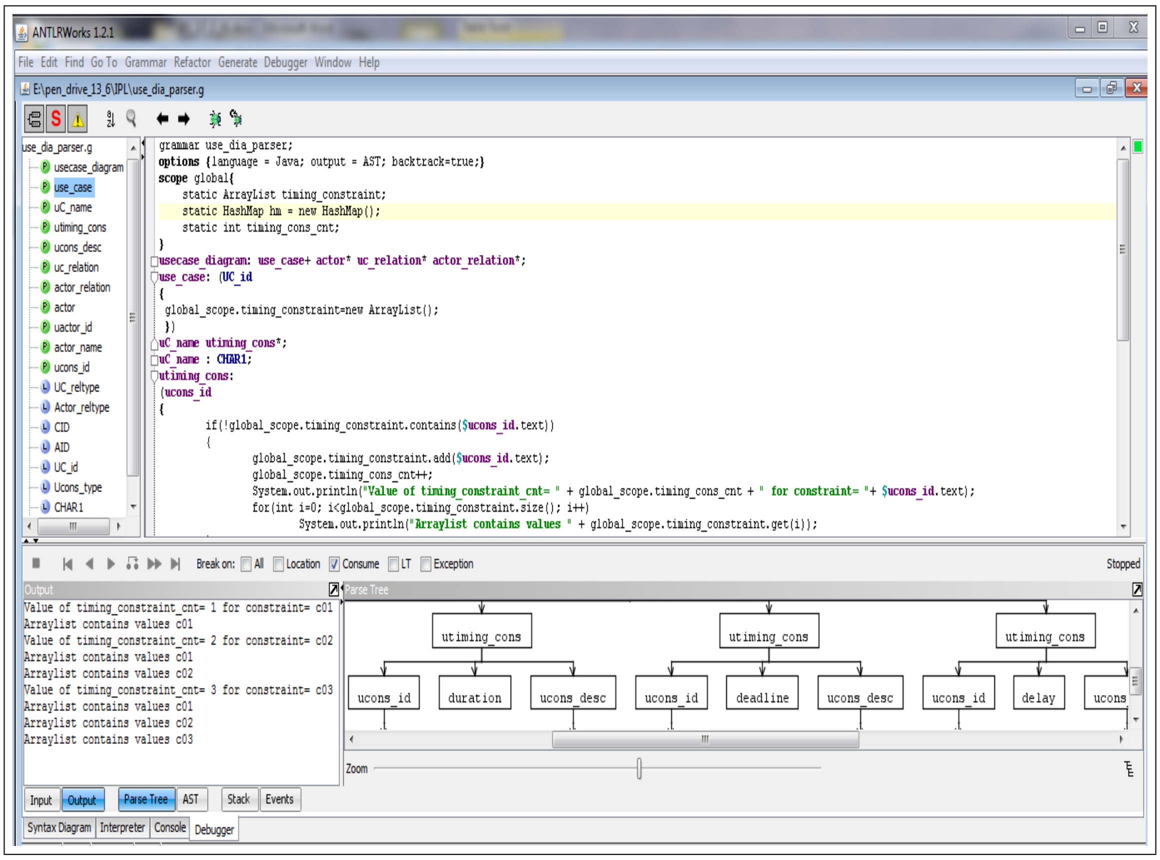
From a formal language description or grammar, ANTLR generates a parser for that language. It builds parse trees which are data structures speaking to how a grammar matches the input. The grammar file for ANTLR contains the lexer and parser rules. After processing the grammar file it generates two classes: lexer and parser. Lexer runs first and splits input into pieces called tokens. Every token speaks to pretty much important bit of information. The stream of tokens is passed to parser which does all necessary works. It is the parser which builds abstract syntax tree, interprets the code or translates it into some other form. Each lexer rule describes one token:

TokenName: regular expression;

Parser rules are more complicated. The most basic version is similar as in lexer rule:

ParserRuleName: regular expression;

An approach for formal specification using ANTLR is provided in [29]. A screenshot of ANTLRWorks is included in Figure 5.1.



```
grammar use_dia_parser;
options {language = Java; output = AST; backtrack=true;}
scope global{
    static ArrayList timing_constraint;
    static HashMap hm = new HashMap();
    static int timing_cons_cnt;
}
usecase_diagram: use_case+ actor* uc_relation* actor_relation*;
use_case: (UC_id
{
 global_scope.timing_constraint=new ArrayList();
 })
uC_name utiming_cons*;
uC_name : CHAR1;
utiming_cons:
(ucons_id
{
        if(!global_scope.timing_constraint.contains($ucons_id.text))
        {
                global_scope.timing_constraint.add($ucons_id.text);
                global_scope.timing_cons_cnt++;
                System.out.println("Value of timing_constraint_cnt= " + global_scope.timing_cons_cnt + " for constraint= "+ $ucons_id.text);
                for(int i=0; i<global_scope.timing_constraint.size(); i++)
                        System.out.println("Arraylist contains values " + global_scope.timing_constraint.get(i));
```

```
Value of timing_constraint_cnt= 1 for constraint= c01
Arraylist contains values c01
Value of timing_constraint_cnt= 2 for constraint= c02
Arraylist contains values c01
Arraylist contains values c02
Value of timing_constraint_cnt= 3 for constraint= c03
Arraylist contains values c01
Arraylist contains values c02
Arraylist contains values c03
```

FIGURE 5.1: Screen shot of ANTLRWorks

Advantages of ANTLR over other lexer/parser generator

- ANTLR generates code in various programming languages such as Ada95, Action Script, C, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby, and Standard ML. But at present Java and C# are the main targets of the current released versions.
- ANTLR (a recursive descent parser) implements a predictive LL(k) parsing strategy to make arbitrary look-ahead to remove conflicts in case of ambiguous grammars.
- One of the advantages of ANTLR is that a “single consistent notation” is used to specify lexers, parsers and tree parsers in ANTLR. This makes ANTLR easier to use.

- The greatest advantage of using ANTLR is that it provides a way to design tools as per user requirements by encoding against each parse rules separately.
- After successful compilation of the formal specification, ANTLR generates a method (for each parse rule) that can be reused during coding for tool design. Developers can also insert their code inside the generated methods.
- There is a provision in ANTLR that one can import a grammar inside another grammar. This can be very helpful for the design of simulator to provide the tool support.

5.3.2 Formal representation: constructs and significance

5.3.2.1 Formal representation of Use case diagram

The parser and the lexer part of the grammar are given in Appendix A (Table A.1) and are also explained here.

Explanation of the grammar

Formal representation starts with the following production rule:
usecase_diagram:(use_case_dia_id { //Unique id for each Use case diagram
global_scope.usecase=new ArrayList(); //List for storing use cases
}) use_case$^+$ actor* uc_relation* actor_relation*;
This denotes that the Use case diagram consists of a unique alphanumeric id (use_case_dia_id), at least one use cases, zero or more actors. These actors and use cases may be related to each others by different UML relations like association, generalization etc.

Use cases are formally represented by the following production rule:
usecase :UC_id {
global_scope.str=$UC_id.text; //storing use case ids
global_scope.uactor=new ArrayList(); //List for storing actors of each use case

global_scope.ustereo=new ArrayList(); //List for storing stereotypes of each use case

global_scope.timing_constraint=new ArrayList();

//List for storing timing constraints of each use case

global_scope.hm = new HashMap();

//Storing all the actors, stereotypes and timing constraints for all use cases

if(!global_scope.usecase.contains($UC_id.text)) //Checking uniqueness of use cases

global_scope.usecase.add($UC_id.text); //Adding use cases in the list

})uc_name uannotation;

Each use case must have a unique alphanumeric use case id (UC_id) associated with its name (uc_name). Different lists are maintained for actors, stereotypes with tag values and timing constraints corresponding to each UC_id for consistency mapping.

UML extensibility mechanisms such as stereotypes with tag values and timing constraints of a Use case diagram can be represented by the following production rule:

uannotation: ustereotype* utiming_cons*;

An annotation represents the following elements:

(i) zero or more stereotypes

(ii) zero or more timing constraints

Semantic actions for consistency checking of actors, stereotypes and timing constraints are incorporated by the following production rules.

ucactor : (uactor_id {

if(!global_scope.uactor.contains($actor_id.text)) //Checking uniqueness of actors

global_scope.uactor.add($actor_id.text); //Adding actors in list

global_scope.hm.put(global_scope.str+"_AT",global_scope.uactor);

//Associates a use case id with its actors

})uactor_name;

utiming_cons: (ucons_id {

if(!global_scope.timing_constraint.contains($ucons_id.text))

//Checking uniqueness of timing constraints

global_scope.timing_constraint.add($ucons_id.text);

//Adding timing constraints in list

global_scope.hm.put($uc_id.text, global_scope.timing_constraint);

//Associates a use case id with its timing constraints

})ucon_type ucons_desc;

ucons_type : 'delay'|'duration'|'deadline';

ustereotype : (ustreo_id {

if(!global_scope.ustereo.contains($ustereo_id.text))

//Checking uniqueness of stereotypes

global_scope.ustereo.add($ustereo_id.text); //Adding stereotypes in list

global_scope.hm.put($uc_id.text, global_scope.ustereo);

//Associates a use case id with its stereotypes

}) stereotype_name '{'utag$^+$'}';

Semantic actions help to identify the uniqueness of each actor, timing constraint and stereotype with tag values. Each time the non terminals uactor_id, ucons_id and ustreo_id are executed, values of the tokens get stored in the respective globally defined Lists namely "uactor", "timing_constraint" and "ustereo". This information will be used further for checking consistency in the next section.

5.3.2.2 Formal specification of Sequence diagram

This section represents consistency checking between Use case and Sequence diagrams with the help of information collected from the previous section. The parser and the lexer part of the grammar are given in Appendix A (Table A.2) and are explained here.

Explanation of the grammar

It starts with the following production rule:

sequence_diagram: (seq_id{

global_scope.seq_diagram=$seq_id.text; //Storing sequence diagram id

global_scope.seq_usecase=new ArrayList(); //Lists for storing usecase id

global_scope.seq_actor=new ArrayList(); //Lists for storing actors

global_scope.seq_stereo=new ArrayList(); //Lists for storing stereotypes

global_scope.seq_timing_constraint= new ArrayList();}

//Lists for storing timing constraints

global_scope.seq_lifeline= new ArrayList(); //Lists for storing lifelines

global_scope.seq_par= new ArrayList(); //Lists for storing participants

global_scope.seq_method= new ArrayList(); //Lists for storing methods

global_scope.seq_message= new ArrayList(); //Lists for storing messages

(uc_id {

if(global_scope.usecase.contains($uc_id.text))

//Try to find out if the current token exists in use case

global_scope.seq_usecase.add($uc_id.text); //Adding use case id in list

}) life_line$^+$ seq_annotation;

This denotes that the Sequence diagram consists of a unique sequence id (seq_id), unique use case id (uc_id), one or more life_lines and annotations. Sequence id is a unique alpha-numeric which points to this Sequence diagram. Eight lists get created for storing use case id, actors, stereotypes with tag values, timing constraints, lifelines, participants, methods and messages respectively. Use case id refers to the Use Case from which the Sequence diagram is realized. The non terminal uc_id captures use case id of a use case from which the sequence diagram is realized.

Each time this rule gets executed, an automated semantic action is performed to check if the use case id in the Sequence diagram is already present in the respective Use case diagram. If it is a success, the use case id is added to the array list "seq_usecase".

Finally, Rule 1 satisfies if the array list ”seq_usecase” contains at least one value which is also present in the array list ”usecase” in the Use case diagram.

A lifeline in the Sequence diagram represents either an actor or a participant. Actors are represented by the following production rule:

actor : (actor_id {
if(global_scope.uactor.contains($actor_id.text))
//Try to find out if the current token exists in use case
if(!global_scope.seq_actor.contains($actor_id.text))
//Checking uniqueness of actors
global_scope.seq_actor.add($actor_id.text); //Adding actors in list
}) actor_name edge$^+$;

The non terminal actor_id captures those actors of a use case which are also present in the corresponding Sequence diagram. Each time this rule gets executed, an automated semantic action is performed to check if the actors in the Sequence diagram are associated with the corresponding use case. If it is a success, the actor id is added to the array list ”seq_actor”.

Finally, Rule 2 satisfies if the contents of both the array lists ”uactor” in the Use case diagram and ”seq_actor” in the Sequence diagram remain same.

Participants are represented by the following production rule:

participant : (par_id {
if(!global_scope.seq_par.contains($par_id.text))
//Checking uniqueness of participants
global_scope.seq_par.add($par_id.text);//Adding participants in list
})par_name edge+;
seq_lifeline.addAll(seq_actor);
seq_lifeline.addAll(seq_par);

Each time the non terminal lifeline is executed, globally defined Lists namely seq_lifeline stores all the actors and participants present in the sequence diagram.

Messages are represented by the following production rule:

msg_func: (msg_id{

if(!global_scope.seq_message.contains($msg_id.text))

//Checking uniqueness of messages

global_scope. seq_message.add($msg_id.text);// Adding messages in list

})msg_name;

Each time the non terminal msg_id is executed, values of the tokens get stored in the globally defined Lists namely seq_message.

UML extensibility mechanisms of a Sequence diagram can be represented by the following production rule:

seq_annotation: seq_stereotype* seq_timing_cons*;

The production rule corresponding to the seq_stereotype is as following:

seq_stereotype:(seq_streo_id {

if(global_scope.ustereo.contains($seq_stereo_id.text))

//Try to find out if the current token exists in use case

if(!global_scope.seq_stereo.contains($seq_stereo_id.text))

//Checking uniqueness of stereotypes

global_scope.seq_stereo.add($seq_stereo_id.text);

//Adding stereotypes in list

}) stereotype_name '{'seq_tag^{+}**'}';**

Each time this rule gets executed, an automated semantic action is performed to check if the stereotype id in the Sequence diagram are associated with the corresponding use case. If it is a success, the stereotype id is added to the array list "seq_stereo".

Finally, Rule 3 satisfies if the contents of both the array lists "seq_stereo" in the Sequence diagram and "ustereo" in the Use case diagram remain same.

The production rule corresponding to the seq_timing_cons is as following:

seq_timing_cons: (seq_cons_id {

if(global_scope.timing_constraint.contains($seq_con_id.text))
//Try to find out if the current token exists in use case
if(!global_scope.seq_timing_constraint.contains($seq_cons_id.text))
//Checking uniqueness of timing constraints
global_scope.seq_timing_constraint.add($seq_cons_id.text);
//Adding timing constraints in list
}) seq_type seq_desc;
seq_type : 'delay'|'duration'|'deadline';
Each time this rule gets executed, an automated semantic action is performed to check if the timing_constraint id in the Sequence diagram associated with the corresponding use case. If it is a success, the timing_constraint id is added to the array list "seq_timing_constraint".

Finally, Rule 4 satisfies if the contents of both the array lists "seq_timing_constraint" in the Sequence diagram and "timing_constraint" in the Use case diagram remain same.

5.3.2.3 Formal specification of Timing diagram

The parser and the lexer part of the grammar are given in Appendix A (Table (A.3)) and are explained here.

Explanation of the Grammar

Formal representation of Timing diagram starts with the following production rule:
timing_diagram: (time_id {
static String timing_diagram=$time_id.text; //Storing timing diagram id
static String t_seq_dia; //storing sequence diagram id
global_scope.par=new ArrayList(); //List for storing participants
global_scope.par_method=new ArrayList(); //List for storing methods
global_scope.par_message=new ArrayList(); //List for storing messages

global_scope.stereo=new ArrayList(); //List for storing stereotypes
global_scope.timing_constraint=new ArrayList();}
//List for storing timing constraints
(seq_id { if(global_scope.seq_diagram.equals($seq_id.text))
//Checking if the current token exists in sequence diagram
global_scope.t_seq_dia=$seq_id.text;//Adding sequence diagram id in the list
})participant$^+$ time annotation;
This denotes that the Timing diagram consists of a unique id, at least one participant, timing parameter and annotations. Timing id is a unique alpha-numeric which points to this Timing diagram. Two String objects and four lists get created corresponding to each Timing diagram for storing Timing diagram id, participants, methods corresponding to each participants, stereotypes with tag values and timing_constraints respectively. Sequence diagram id refers to the sequence diagram from which the timing diagram is realized.
Each time this rule gets executed, an automated semantic action is performed to check if the sequence diagram id in the Timing diagram is already present in the respective Sequence diagram. If it is a success, the sequence diagram id is added to the list "t_seq_dia".

Finally, Rule 5 satisfies if both the lists "t_seq_dia" and "seq_diagram" contain same value.

Participants are represented by the following production rule:
participant: (par_id{
if(global_scope.seq_lifeline.contains($par_id.text))
//Checking if the current token exists in sequence diagram
if(!global_scope.par.contains($par_id.text))
//Checking uniqueness of participants
global_scope.par.add($par_id.text); //Adding participants in the list
}par_name state$^+$ event$^+$;
The non terminal par_id captures those lifelines of a sequence diagram which are also

present as participants in the corresponding timing diagram. Each time this rule gets executed, an automated semantic action is performed to check if the participants in the timing diagram are associated with the respective sequence diagram. If it is a success, the participant's id is added to the array list "par".

Finally, Rule 6 satisfies if the contents of both the array lists "seq_lifeline" in the Sequence diagram and "par" in the Timing diagram remain same.

Messages are represented by the following production rule:
msg : (msg_id {
if(global_scope.seq_message.contains($msg_id.text))
//Checking if the current token exists in sequence diagram
if(!global_scope.par_message.contains($msg_id.text))
//Checking uniqueness of messages
global_scope.par_message.add($msg_id.text); //Adding messages in the list
}) mgs_name;
Each time this rule gets executed, an automated semantic action is performed to check if the messages in the timing diagram are associated with the respective sequence diagram. If it is a success, the message id is added to the array list "par_message".

Finally, Rule 7 satisfies if the contents of both the array lists "seq_message" in the Sequence diagram and "par_message" in the Timing diagram remain same.

Methods are represented by the following production rule:
mthd : (mthd_id {
if(global_scope.seq_method.contains($mthd_id.text))
//Checking if the current token exists in sequence diagram
if(!global_scope.par_method.contains($mthd_id.text))
//Checking uniqueness of methods
global_scope.par_method.add($mthd_id.text); //Adding methods in the list

}) mthd_name;

Each time this rule gets executed, an automated semantic action is performed to check if the methods in the timing diagram are associated with the respective sequence diagram. If it is a success, the method id is added to the array list "par_method".

Finally, Rule 8 satisfies if the contents of both the array lists "seq_method" in the Sequence diagram and "par_method" in the Timing diagram remain same.
UML extensibility mechanisms of a Timing diagram can be represented in similar way as described in the previous section.

The production rule corresponding to the stereotype is as following:
stereotype : (streo_id {
if(global_scope.seq_stereo.contains($stereo_id.text))
//Checking if the current token exists in sequence diagram
if(!global_scope.stereo.contains($stereo_id.text))
//Checking uniqueness of stereotypes
global_scope.stereo.add($stereo_id.text); //Adding stereotypes in the list
}) stereotype_name '{'tag^{+}'}';
Each time this rule gets executed, an automated semantic action is performed to check if the stereotype id in the Timing diagram are associated with the corresponding Sequence diagram. If it is a success, the stereotype id is added to the array list "stereo".

Finally, Rule 9 satisfies if the contents of both the array lists "seq_stereo" in the Sequence diagram and "stereo" in the Timing diagram remain same.

Timing_constraints are represented by the following production rule:
timing_cons: (cons_id {
if(global_scope.seq_timing_constraint.contains($cons_id.text))
//Checking if the current token exists in sequence diagram
if(!global_scope.timing_constraint.contains($cons_id.text))

//Checking uniqueness of timing constraints

global_scope.timing_constraint.add($cons_id.text);

//Adding timing constraints in the list

}) cons_type cons_desc;

Each time this rule gets executed, an automated semantic action is performed to check if the timing_constraint id in the Timing diagram associated with the corresponding Sequence diagram. If it is a success, the timing_constraint id is added to the array list "timing_constraint".

Finally, Rule 10 satisfies if the contents of both the array lists "seq timing constraint" in the Sequence diagram and "timing _constraint" in the Timing diagram remain same.

A good design is such that all the lifelines, methods, messages, stereotypes with tag values and timing constraints related to a sequence diagram are also present in the corresponding timing diagram without losing any important information. Each time the rules corresponding to the non terminals par_id, mthd_id, msg_id, streo_id and cons_id get executed, automated checking is performed whether the respective lifelines, methods, messages, stereotypes with tag values and timing constraints are already present in the sequence diagram.

5.3.2.4 Formal specification of Class diagram

This section represents consistency checking between Timing and Class diagrams with the help of information collected from the previous section. The parser and the lexer part of the grammar are given in Appendix section (Table A.4) and are explained here.

Explanation of the Grammar

Formal representation of Class diagram starts with the following production rule:

class_diagram: (class_dia_id { //Unique id for Class diagram

global_scope.class=new ArrayList(); //List for storing classes

global_scope.c_method=new ArrayList(); //List for storing methods

global_scope.c_stereo=new ArrayList(); //List for storing stereotypes

global_scope.c_tim_con=new ArrayList(); //List for storing timing constraints

})

(td_id{

if(global_scope.timing_diagram.equals($td_id.text))

//Try to find out if the current token exists in Timing diagram

global_scope.c_timing_diagram=$td_id.text; //Adding Timing diagram id

})classes+ relation*;

This denotes that each Class diagram consists of a unique alpha-numeric Class diagram id, Timing diagram id, one or more classes and zero or more relations. Four lists get created corresponding to each Class diagram for storing classes, methods, stereotypes with tag values and timing constraints respectively. Classes are related to each other by different UML relations like association, generalization etc. Timing diagram id refers to the Timing diagram from which the Class diagram is realized.

Classes are represented by the following production rule:

classes: (class_id {

if(global_scope.par.contains($class_id.text))

//Try to find out if the current token exists in Timing diagram

if(!global_scope.class.contains($class_id.text)) //Checking uniqueness of classes

global_scope.class.add($class_id.text);//Adding classes in list

})cname method_class* attribute* c_annotation;

Each time this rule gets executed, an automated semantic action is performed to check if the class id in the Class diagram exists as a participant in the respective Timing diagram. If it is a success, the class id is added to the array list "class".

Finally, Rule 11 satisfies if the contents of both the array lists "par" in the Timing diagram and "class" in the Class diagram remain same.

Methods are represented by the following production rule:

method_class:(method_id{

if(global_scope.par_method.contains($method_id.text))

//Try to find out if the current token exists in Timing diagram

if(!global_scope.c_method.contains($method_id.text))

//Checking uniqueness of methods

global_scope.c_method.add($method_id.text); Adding methods in list

})Access_specifier Data type method name '('parameter list')' c annotation;

Each time this rule gets executed, an automated semantic action is performed to check if the method id in the Class diagram exists in the respective Timing diagram. If it is a success, the method id is added to the array list "c method".

Finally, Rule 12 satisfies if the contents of both the array lists "par_method" in the Timing diagram and "c_method" in the Class diagram remain same.

Annotation can be represented by the following production rule:

c_annotation: c_stereotype* c_timing_cons*;

An annotation represents the following elements:

i) zero or more stereotypes

ii) zero or more timing constraints

The production rule corresponding to the stereotype is as following:

c_stereotype : (c_streo_id{

if(global_scope.stereo.contains($c_stereo_id.text))

//Try to find out if the current token exists in Timing diagram

if(!global_scope.c_stereo.contains($c_stereo_id.text))

//Checking uniqueness of stereotypes

global_scope.c_stereo.add($c_stereo_id.text); Adding stereotypes in list

}) stereotype_name '{'c_tag^{+}**'}';**

Each time this rule gets executed, an automated semantic action is performed to check if the stereotype id in the Class diagram exists in the respective Timing diagram. If it is a success, the stereotype id is added to the array list "c_stereo".

Finally, Rule 13 satisfies if the contents of both the array lists "stereo" in the Timing diagram and "c_stereo" in the Class diagram remain same.

The production rule corresponding to the timing_constraint is as following:
c_timing_cons: (c_cons_id{
if(global_scope.timing_constraint.contains($c_cons_id.text))
//Try to find out if the current token exists in Timing diagram
if(!global_scope.c_tim_con.contains($c_cons_id.text))
//Checking uniqueness of timing constraints
global_scope.c_tim_con.add($c_cons_id.text); //Adding timing constraints in list
}) c_type c_desc;
Each time this rule gets executed, an automated semantic action is performed to check if the timing_constraint id in the Class diagram exists in the respective Timing diagram. If it is a success, the timing_constraint id is added to the array list "c_tim_con".

Finally, Rule 14 satisfies if the contents of both the array lists "timing_constraint" in the Timing diagram and "c_tim_con" in the Class diagram remain same.

A good design is such that all the classes, methods, stereotypes with tag values and timing_constraints related to a timing_diagram are also present in the corresponding Class diagram without losing any important information.

5.4 Case Study

The case study described in Chapter 4 has been adopted for applying our methodologies and substantiation of our approaches. Ventilation support is routinely needed for critically ill adults in intensive care units. This process can decrease the patient's work of breathing by unloading respiratory muscles in a synchronous manner. Physicians must be knowledgeable about the function and limitations of ventilator modes, causes of respiratory distress and dyssynchrony with the ventilator, and appropriate management to provide high-quality patient-centered care.

The following section represents the automated consistency checking among the mentioned UML diagrams (Figure 4.7, Figure 4.8, Figure 4.9 and Figure 4.10) based on the rules in section 5.2.1. Table 5.2, 5.3 and 5.4 represent information about the use cases, stereotypes and timing constraints respectively.

5.5 Automated checking of inter diagram consistency

Figure 5.2 represents the block diagram of the working methodology.

Most of the modeling tools (Rational Rose, Magic draw, Altova etc.) provide the facilities to generate XMI files from UML diagrams. We have developed a simulator tool which performs the following tasks:

(a) Parses the XMI files (developed from UML diagrams) and accordingly generates the set of intermediate tables (such as Table 5.2, 5.3 and 5.4)

(b) Builds input strings respectively from these tables for the grammar in sections 5.3.2.1, 5.3.2.2, 5.3.2.3 and 5.3.2.4

(c) Executes the formal grammars and also the code embedded within the methods of the formal grammars

(d) Generates various consistency verification Table (such as Table 5.5, Table 5.6)

(e) Provides decision of inter diagram consistency or inconsistency

Table 5.2: Use case information

Use_Case	**UID**
Deliver Ventilation	u01
Alarm on Critical Events	u02
Informational Alarm	u03
Caution Alarm	u04
Critical Alarm	u05
Set Ventilation Parameter	u06
Monitor Breathing Circuit	u07
Set Inspiratory Flow	u08
Set Tidal Volume	u09
Set Inspiratory Pause	u10
Set I:E Ratio	u11
Set Inspiratory Time	u12
Set Respiration rate	u13
Set maximum pressure limit	u14

Table 5.3: Stereotype information

Stereo_Name	**STID**
<<inspiratoryFlow>>	st01
<<tidalVolume>>	st02
<<inspiratoryPause>>	st03
<<I:ERatio>>	st04
<<inspiratoryTime>>	st05
<<respirationRate>>	st06
<<maxPressureLimit>>	st07

Table 5.4: Timing constraint information

TCID	**Timing_cons_Type**
c01	Deadline
c02	Duration
c03	Delay

5.6 Results and discussion

Based on the Use case diagram in Figure 4.7 it is observed that the use case u06 includes u08, u09, u10, u11, u12, u13 and u14; hence, the annotations belong to those use cases may be considered as part of u06.

Our application automatically develops the following valid input string for the grammar presented in section 5.3.2.1.

In order to make it simple, the following input string is formed by considering only three use cases (u01, u02 and u03).

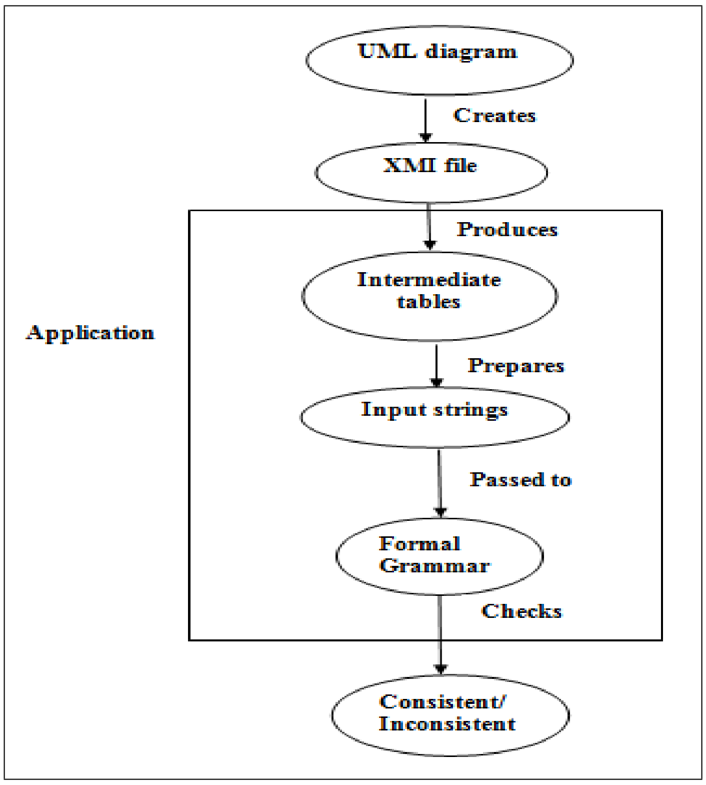
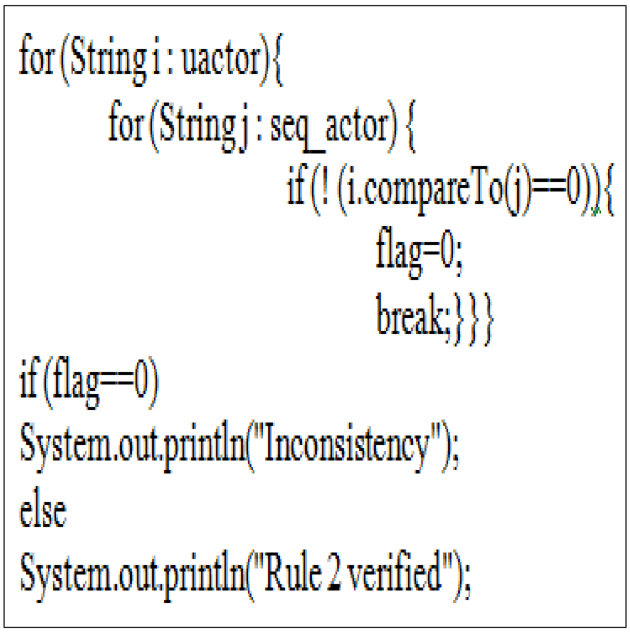


FIGURE 5.2: Block diagram of the automated verification of inter diagram consistency



```
for(String i : uactor){
        for(String j : seq_actor) {
                        if(! (i.compareTo(j)==0)){
                                flag=0;
                                break;}}}
if(flag==0)
System.out.println("Inconsistency");
else
System.out.println("Rule 2 verified");
```

FIGURE 5.3: Code snippet for verification

ud01 u01 Deliver_Ventilation u06 Set_Ventilation_Parameter c01 deadline 12_ventilations_per_minutes u08 Set_Inspiratory_Flow st01 <<inspiratoryFlow>> {initialValue (1, L/min) finalValue (180, L/min) defaultValue (100, L/min)} a01 patient a02 physician u06 <<include>> u08.

In this dissertation, only the Sequence diagram corresponding to the use case u06 has been considered.

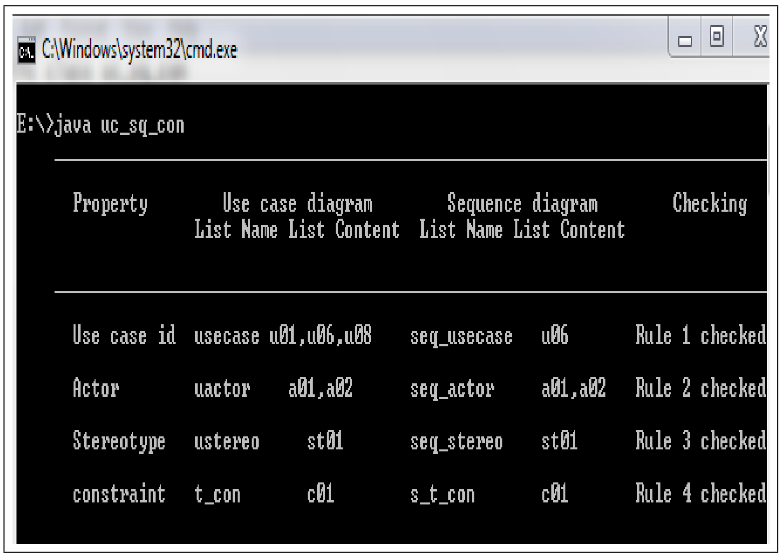
Similarly, a set of intermediate tables can be generated from the Sequence diagram (Figure 4.8) with id s01 corresponding to the use case u06. The following valid input string is developed by our application for the grammar presented in section 5.3.2.2.

u06 s01 a01 patient a02 physician E01 event1 send 1 MS01 Set_Inspiratory_Flow E02 event2 recv 2 st01 <<inspiratoryFlow>> {initialValue (1, L/min) finalValue (180, L/min) defaultValue (100, L/min)} tc01 deadline (d1 + d2 $\leq$ 5).

Similarly, a set of intermediate tables can be generated from the Timing diagram (Figure 4.9). The simulator tool automatically develops the following valid input string for the grammar presented in section 5.3.2.3. In order to make it simple, the following input string is formed by considering only two participants pc01 and pc02.

s01 t01 pc01 Physician s01 idle e01 event1 mt01 Set_Inspiratory_Flow pc01 s01 idle s02 waiting pc02 Parameter_Controller s01 idle e02 event2 mt02 Open_Valve pc02 s01 idle s04 Valve_Opened e03 even3 mt03 Close_Valve pc02 s01 idle s05 Valve_Closed 1 st01 <<inspiratoryFlow>> {initialValue (1, L/min) finalValue (180, L/min) defaultValue (100, L/min)} tc01 deadline desc.

Similarly, a set of intermediate tables can be generated from the Class diagram (Figure 4.10) and the following valid input string is developed by our application for the



```
E:\>java uc_sq_con
     ________________________________________________________________________
     Property      Use case diagram         Sequence diagram        Checking
                List Name List Content   List Name List Content
     ________________________________________________________________________
     Use case id  usecase u01,u06,u08    seq_usecase    u06      Rule 1 checked
     Actor        uactor     a01,a02     seq_actor      a01,a02  Rule 2 checked
     Stereotype   ustereo     st01       seq_stereo     st01     Rule 3 checked
     constraint   t_con       c01        s_t_con        c01      Rule 4 checked
```

FIGURE 5.4: Verification of consistency between Use case and Sequence diagrams

grammar presented in section 5.3.2.4.

c01 t01 pc01 Physician mt01 + void Set_Inspiratory_Flow () st01 <<inspiratoryFlow>> {initialValue (1, L/min) finalValue (180, L/min) defaultValue (100, L/min)} pc02 Parameter_Controller mt02 + void Open_Valve() mt03 + void Close_Valve() tc01 deadline Duration between opened and closed valve must be less than d units Physician 1..2 Parameter_Controller c01 descr association.

5.6.1 Consistency of Use case and Sequence diagrams

After the input string is given to the ANTLR, the system will automatically store the data values in different lists as shown in Figure 5.4.

Verification of Rule 1:

According to Rule 1, each Sequence diagram realizes a use case, which should be present in the Use case diagram.

From Figure 5.4, we observe that the value of the arraylist "seq_usecase" already present in the arraylist "usecase".

Hence, we can conclude that Rule 1 is satisfied.

Verification of Rule 2:

According to Rule 2, actors associated with a use case should occur in the corresponding Sequence diagram.

From Figure 5.4, we observe that the value of both the arraylists "uactor" and "seq_actor" remain same.

After verifying the grammar successfully, ANTLR generates method for each parse rule. In order to verify the inter diagram consistency rules, simulator is used to embed the code within the method actor (generated from parse rule actor of the grammar introduced in section 5.3.2.2). Figure 5.3 shows the code snippet of the method actor for the verification of inter diagram consistency rules.

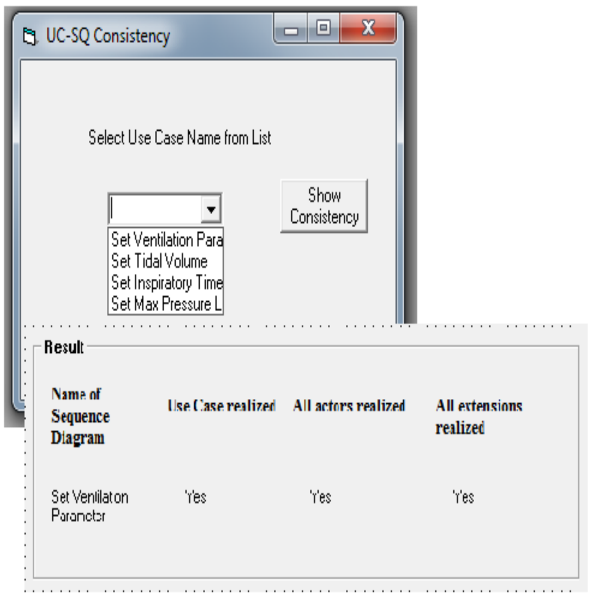
Verification of Rule 3:

According to Rule 3, each stereotype associated with a use case must be present in the corresponding Sequence diagram.

From Figure 5.4, we observe that the value of both the arraylists "ustereo" and "seq_stereo" remain same.

Verification of Rule 4:

According to Rule 4, different timing_constraints associated with a use case must be present in the corresponding Sequence diagram.



Name of Sequence Diagram	Use Case realized	All actors realized	All extensions realized
Set Venilaton Parameter	Yes	Yes	Yes

FIGURE 5.5: Panel for showing consistency between Use case and Sequence diagrams

From Figure 5.4, we observe that the value of both the arraylists "timing_constraint" and "seq_timing_constraint" remain same.

Some sample screen snapshots have been provided in Fig 5.5.

5.6.2 Consistency of Sequence and Timing diagrams

After the input string is given to the ANTLR, the system will automatically store the data values in different lists as shown in Table 5.5.

Verification of Rule 5:

According to Rule 5, any Timing diagram can be realized from only one Sequence diagram.

TABLE 5.5: Verification of consistency between Sequence and Timing diagrams

Property	**Sequence Diagram**		**Timing Diagram**		**Verification**
	Array_List_Name	**Array_List_Value**	**Array_List_Name**	**Array_List_Value**	
Sequence diagram id	Sequence_diagram	s01	t_seq_dia	s01	Rule 5 verified
Participant	seq_lifeline	pc01, pc02	par	pc01, pc02	Rule 6 verified
Message	seq_message	ms01, ms02, ms03	par_message	ms01, ms02, ms03	Rule 7 verified
Method	seq_method	mt01, mt02, mt03	par_method	mt01, mt02, mt03	Rule 8 verified
Stereotype	seq_stereo	st01	stereo	st01	Rule 9 verified
Timing_constraint	seq_timing_constraint	tc01	timing_constraint	tc01	Rule 10 verified

From Table 5.5, we observe that the value of the arraylist ”t_seq_dia” already present in the arraylist ”sequence_diagram”.

Hence, we can conclude that Rule 5 is satisfied.

Verification of Rule 6:

According to Rule 6, each lifeline in the Sequence diagram should be mapped to one participant in the corresponding Timing diagram.

Now, from Table 5.5, we observe that the value of both the array lists ”par” and ”seq_lifeline” remain same.

Hence, we can infer that the lifelines in the Sequence diagram are consistent with the participants in the corresponding Timing diagram.

Verification of Rule 7:

According to Rule 7, each message in the Sequence diagram must be present in the corresponding Timing diagram.

Now, from Table 5.5, we observe that the value of both the array lists ”par_message” and ”seq_message” remain same.

Hence, we can infer that the messages in the Sequence diagram are consistent with the messages in the corresponding Timing diagram.

Verification of Rule 8:

According to Rule 8, each method in the Sequence diagram must be present in the corresponding Timing diagram.

Now, from Table 5.5, we observe that the value of both the array lists "par_method" and "seq_method" remain same.

Hence, we can infer that the methods in the Sequence diagram are consistent with the methods in the corresponding Timing diagram.

Verification of Rule 9:

According to Rule 9, each stereotype in the Sequence diagram must be present in the corresponding Timing diagram.

From Table 5.5, we observe the following:

- number of elements in the array lists "seq_stereo" and "stereo" are equal which is 1
- both the array lists contain the same set of strings

We conclude that the stereotypes in the Sequence diagram are consistent with the stereotypes in the corresponding Timing diagram.

Verification of Rule 10:

According to Rule 10, each timing_constraint in the Sequence diagram must be present in the corresponding Timing diagram.

From Table 5.5, we observe the following:

- number of elements in the array lists "timing_constraint" and "seq_timing_constraint" are equal which is 1
- both the array lists contain the same set of strings

TABLE 5.6: Verification of consistency between Timing and Class diagrams

Property	**Timing Diagram**		**Class Diagram**		**Verification**
	Array_List_Name	**Array_List_Value**	**Array_List_Name**	**Array_List_Value**	
Participant	par	pc01, pc02	class	pc01, pc02	Rule 11 verified
Method	par_method	mt01, mt02, mt03	c_method	mt01, mt02, mt03	Rule 12 verified
Stereotype	stereo	st01	c_stereo	st01	Rule 13 verified
Timing_constraint	timing_constraint	tc01	c_tim_con	tc01	Rule 14 verified

We conclude that the timing_constraints in the Sequence diagram are consistent with the timing_constraints in the corresponding Timing diagram.

5.6.3 Consistency of Timing and Class diagrams

After the input string is given to the ANTLR, the system will automatically store the data values in different lists as shown in Table 5.6.

Verification of Rule 11:

According to Rule 11, each participant in the Timing diagram should be mapped to one class in the Class diagram.

Now, from Table 5.6, we observe the following:

- number of elements in the array lists "par" and "class" are equal which is 2
- both the array lists contain the same set of strings

Hence, we can infer that the participants in the Timing diagram are consistent with the classes in the corresponding Class diagram.

Verification of Rule 12:

According to Rule 12, each method in the Timing diagram must be present in the Class diagram.

Now, from Table 5.6, we observe the following:

- number of elements in the array lists "par_method" and "c_method" are equal which is 3
- both the array lists contain same set of strings

Verification of Rule 13:

According to Rule 13, each stereotype in the Timing diagram must be present in the Class diagram.

From Table 5.6, we observe the following:

- number of elements in the array lists "stereo" and "c_stereo" are equal which is 1
- both the array lists contain the same set of strings

We conclude that the stereotypes in the Timing diagram are consistent with the stereotypes in the corresponding Class diagram.

Verification of Rule 14:

According to Rule 14, each timing_constraint in the Timing diagram must be present in the Class diagram.

From Table 5.6, we observe the following:

- number of elements in the array lists "timing_constraint" and "c_tim_con" are equal which is 1
- both the array lists contain the same set of strings

We conclude that the timing_constraints in the Timing diagram are consistent with the timing_constraints in the corresponding Class diagram.

5.7 Comparison with related work

The existing model checking tools (Spin, CADP, Alloy, FDR2 etc.) [42] require input specifications to be provided in some specific manner, like Promela for Spin, first order logic for Alloy, LOTOS-NT for CADP and CSPm for FDR2 etc. In comparison to that, our framework minimizes this extra overhead. Given a UML model, the framework automatically verifies the consistency rules. In [24], Jinho et al. proposed a systematic approach for checking timing consistency rules among three UML diagrams - state machine, sequence, and timing diagrams using two case studies with MARTE annotations. However, their approach lacks formal verification. In contrast with that, our work presented in this Chapter formally verifies the consistency between use case, sequence, timing and class diagrams.

5.8 Threats to validity of the proposed approach

We have introduced consistency between Use case, Sequence, Timing and Class diagram. Further, a simulator has been developed based on the formal specification to automatically detect inconsistencies at the early stage of design. Most of the modeling tools (Rational Rose, Magic draw, Altova etc.) provide the facilities to generate XMI files from UML diagrams. Different modeling tools generate different formats of XMI files. In this research work, we have used XMI file generate from Magic draw UML. Our simulator tool parses this specific XMI file and builds input strings respectively for the proposed grammars. This simulator tool may need to be modified to handle the XMI file developed from other modeling tools.

5.9 Conclusion

This Chapter develops a framework for consistency checking of the model for real-time patient ventilation system. A software development life cycle consists of several phases such as feasibility study, requirements analysis, design etc. This framework mainly focuses on the requirements analysis and design phase of a software development life cycle. In this chapter, we have defined a set of consistency rules that should be maintained to ensure that the different UML models capturing different yet related aspects of the system are consistent within themselves. UML/MARTE annotated Use case, Sequence, Timing and Class diagrams have been considered and we have first established the relationships among them along with the common element of relationship. Based on this we have defined a Weighted design graph which visually depicts the consistency conditions. Further, formal representation with semantic actions is also developed for the automated checking of consistency rules among these diagrams. A simulator is developed based on the formal specification to automatically detect inconsistencies at the early stage of design.

Chapter 6

Verification of Real-Time Software Systems

6.1 Introduction

Requirements engineering is an approach by which requirements for a software are inspired, reported, dissected and overseen all through the software development life cycle, including operations of tracing requirements [99]. Traceability helps to measure and assess each requirements of a system to make it functional as per the client's needs. This is one of the most important processes which must be executed very carefully. With the help of traceability, we can prove that all requirements have been implemented correctly. Requirements can be validated into design phase if they have been traced to design elements.

As Real-Time Systems depend on events under some timing constraints, the traceability of RTSS must be done completely. In this dissertation, we mainly concentrate on the traceability of real-time requirements with an emphasis on timing constraints. Requirements tracing helps to verify if all software requirements have been evolved to design, code and test cases. This chapter aims at proposing a metric based requirement traceability framework for RTSS. Metrics, as indicators, provide a quantitative measurement of RTSS and focus on problem areas in the systems. The more the degree of the traceability completeness, the less will be the reduced defect rate and required efforts and also enhances the quality for a developed software [74, 103].

In this chapter, we have developed a new metrics suite named Requirement Coverage Metrics for Real-Time (RCM-RT) which traces the timing constraints from requirements into use cases, use cases into sequence diagrams, sequence diagrams into timing diagrams and timing diagrams into class diagrams. RCM-RT measures the extent to which requirements have been realized and implemented subsequently in the analysis and design phases. The quantitative assessment also helps in judging the quality of the system with respect to its realization of requirements. Further, we have presented formal specification language to capture the requirements traceability of timing constraints among different UML diagrams. A simulator tool is developed that automatically generates the traceability metrics for timing constraints from a given set of requirements and UML design models.

Formal methods are intended to systematize and bring thoroughness at every phase of software development. This helps us to avoid overlooking critical issues, gives a standard intends to record different suppositions and choices and forms a basis for consistency among many related activities. Our approach will facilitate software designers, project managers and architects to automatically generate the traceability metrics of real-time requirements at the early stage of design.

6.2 Requirements Traceability

RCM-RT will be useful in requirement management as well as project management of real-time software projects. It helps to detect missing timing requirements.

The developed metrics suite (RCM-RT) comprises of the following sets of metrics :

- RUC-RT (Requirements - Use Case coverage for Real-Time),
- USC-RT (Use Case diagram - Sequence diagram coverage for Real-Time),
- STC-RT (Sequence diagram - Timing diagram coverage for Real-Time),
- TCC-RT (Timing diagram - Class diagram coverage for Real-Time) and
- RCF-RT (Requirement Coverage Factor for Real-Time) that measures the average value of all the traceability metrics for a specific requirement.

The details of RCM-RT are explained in the following section.

RCM-RT

U_R: Set of unique use cases in use case diagram with respect to a specific requirement.

U_R={u_i|u_i∈U, U∈UC }

TC_U: Set of unique timing constraints associated with a use case.

TC_U={tc_i|tc_i∈TC}....................(1)

S_U: Sequence diagram corresponding to a particular use case u_i, u_i *in* U_R (We consider one to one between use case and sequence diagram).

S_U={sq_i|sq_i∈SQ}

TC_S: Set of timing constraints associated with the sequence diagram.

TC_S = {tc_i|tc_i∈TC }..............(2)

An empty set TC_S signifies that no timing constraints present in the sequence diagram.

TC_{U-S} : Set of timing constraints traced from the use case to the sequence diagram.

TC_{U-S} = {tc_i|tc_i∈TC_S and tc_i∈TC_U}...........(3)

T_S: Timing diagram corresponding to the sequence diagram sq_i, $sq_i \in S_U$.

TC_T: Set of timing constraints defined in the timing diagram T_S.

TC_T: $= \{tc_j | tc_j \in \mathrm{TC}\}$............(4)

An empty set TC_T signifies that no timing constraints have been defined in the timing diagram T_S.

TC_{S-T}: Set of timing constraints traced from the sequence diagram sq_i, $sq_i \in S_U$ to the timing diagram T_S.

$TC_{S-T} = \{tc_i | tc_i \in TC_S$ and $tc_i \in TC_T\}$............(5)

C_T: The class diagram corresponding to the timing diagram T_S.

TC_C: Set of timing constraints defined in the class diagram C_T.

$TC_C = \{tc_j | tc_j \in \mathrm{TC}\}$

An empty set TC_C signifies that no timing constraints have been defined in the class diagram C_T.

TC_{T-C}: Set of timing constraints traced from the timing diagram T_S to the class diagram C_T.

$TC_{T-C} = \{tc_i | tc_i \in TC_C$ and $tc_i \in TC_T\}$...........(6)

1. RUC-RT: This signifies whether there exists at least one use case for each requirement. In that case, the value of RUC-RT will be 1 otherwise 0.

   This calculates trace of requirements into use cases.

2. USC-RT: It is the ratio of the number of timing constraints traced from a use case to its corresponding sequence diagram, to the total number of timing constraints present in that use case.

   USC-RT = N(TC_{U-S})/N(TC_U) ....(7)

This calculates the trace of timing constraints from use cases to sequence diagrams.

3. STC-RT: It is the ratio of the number of timing constraints traced from a sequence diagram to the respective timing diagram, to the total number of timing constraints present in that sequence diagram.

   STC-RT = N(TC_{S-T})/N(TC_S)................(8)

   This calculates the trace of timing constraints of sequence diagrams into timing diagrams.

4. TCC-RT: It is the ratio of the number of timing constraints traced from a timing diagram to its corresponding class diagram, to the total number of timing constraints present in that timing diagram.

   TCC-RT = N(TC_{T-C})/N(TC_T))...............(9)

   This calculates the trace of timing constraints of timing diagrams into class diagrams.

5. RCF-RT: It is defined as the average value of RUC-RT, USC-RT, STC-RT and TCC-RT.

   $$\text{RCF-RT} = \frac{\text{RUC-RT + USC-RT + STC-RT + TCC-RT}}{4} \text{............(10)}$$

   The value of RCF-RT depends upon the requirement coverage in each of the phases of its implementation in use case, sequence, timing and class diagram. Thus 0 $\leq$ RCF-RT $\leq$ 1.

RCM-RT can be defined as follows.

$$\text{RCM-RT} = \frac{\text{Sum of RCF-RT for all requirements}}{\text{Total number of requirements}}$$

$$= \frac{\sum_{Rid=1}^{n} RCF - RT}{N(R)}$$

where Rid = Requirement id.

Thus RCM-RT gives a quantitative measurement of extent of requirement coverage in design.

6.3 Formal specification for the automatic generation of traceability metrics

In this research work, we consider formal grammar of four UML diagrams - Use case, Sequence, Timing and Class diagrams for the automatic generation of traceability metrics. Semantic actions in the formal approach automatically generates traceability metrics at the early stage of the design. For the implementation of the traceability metrics from these UML diagrams, we choose ANTLR [95] which is a translator generator tool whose detailed specification is already defined in Chapter 5.

6.3.1 Formal specification to calculate N(TC_U) from Use case diagram

Partial snapshot of the grammar for the automated generation of N(TC_U) is shown in Table 6.1.

Table 6.1: Formal grammar for the automated generation of N(TC_U) from Use case diagram

```
grammar use_dia_parser;
options{language = Java; output = AST; backtrack=true;}
scope global{
static ArrayList timing_constraint; //TC_U in rule(1)
static HashMap hm; //Storing timing constraints for all use cases
static HashMap hm_ruc =new HashMap();
/*Storing requirement corresponding to each use case*/
static int timing_cons_cnt;       //N(TC_U)
usecase_diagram: use_case+ actor* uc_relation* actor_relation*;........................(a)
use_case: (Req_id uc_id {
global_scope.timing_constraint= new ArrayList();
//Storing timing constraints of each use case
   global_scope.hm = new HashMap();
global_scope.hm_ruc.put($Req_id.text,$uc_id.text);
}) uc_name utiming_cons*;.............................(b)
utiming_cons:(ucons_id{//Semantic actions.......(c)
if(!global_scope.timing_constraint.contains
($ucons_id.text)) {//Checking uniqueness of TC_U
global_scope.timing_constraint.add ($ucons_id.text);
//Adding timing constraints in TC_U
global_scope.timing_cons_cnt++; //N(TC_U)=N(TC_U)+1
global_scope.hm.put($uc_id.text,
global_scope.timing_constraint);
................
```

Explanation of the grammar

Formal grammar starts with the production rule usecase_diagram (a) which consists of one or more use cases, zero or more actors, zero or more relations between use cases and zero or more relations between actors.

Each use case (rule use_case (b)) must contain unique alphanumeric requirement id (Req_id), use case id (uc_id) associated with a name (uc_name) and timing constraints (utiming_cons). Each timing constraint (rule (c)) is represented by the unique id (ucons_id) which is checked semantically. Each time the non-terminal ucons_id is executed, the value of the token gets stored in the globally defined

ArrayList timing_constraint and subsequently the count of timing constraint timing_cons_cnt gets incremented by one.

Hashmap hm_ruc and hm record every <requirement, use case> and <use case, timing_constraint> pair respectively. On successful execution, the value of N(TC_U) will show the actual number of timing constraints present in the diagram.

6.3.2 Automated calculation of N(TC_S) and N(TC_{U-S}) from Sequence diagram

Partial snapshot of the grammar is shown in Table 6.2.

Explanation of the grammar

The grammar starts with the production rule sequence_diagram (d) which consists of a unique use case id (uc_id), unique sequence id (seq_id), one or more lifelines and zero or more timing constraints(seq_timing_cons as in rule (e)). Use case id refers to the use case from which the Sequence diagram is realized. Sequence id is a unique alpha-numeric which refers to this Sequence diagram.

The non-terminal seq_cons_id is responsible for tracing the flow of timing constraints from a use case to the respective Sequence diagram. Each time this rule gets executed, an automated checking is performed against the timing constraints already present in the use case. Subsequently, the count of timing constraint use_timing_constraint_cnt gets incremented by one. This counter keeps track of the number of timing constraints present in both the use case and the corresponding Sequence diagram.

If the final value of N(TC_U) is equal to the value of N(TC_{U-S}) we can infer that all the timing constraints in the use case have been traced in the corresponding Sequence diagram.

Table 6.2: Formal grammar for the automated generation of N(TC_S) and N(TC_{U-S}) from Sequence diagram

```
grammar seq_dia;
options{ language = Java; output=AST; backtrack=true;
tokenVocab = use_dia_parser;}
scope global{
static ArrayList seq_timing_constraint; //TC_S in rule (2)
static int use_timing_constraint_cnt;       //N(TC_U-S)
static int seq_timing_constraint_cnt; }   //N(TC_S)
sequence_diagram:uc_id (seq_id{ //semantic action
global_scope.seq_timing_constraint=new ArrayList();
//Storing timing constraints of sequence diagram
})life_line+ seq_timing_cons* .....................(d);
.....................
seq_timing_cons: (seq_cons_id{ //Semantic actions
if(global_scope.timing_constraint.contains ($seq_cons_id.text))
//Try to find out if the current token exists in TC_U
global_scope.use_timing_constraint_cnt++; //N(TC_U-S)= N(TC_U-S)+1
if(!global_scope.seq_timing_constraint.contains ($seq_con_id.text)){
//Checking uniqueness in TC_S
global_scope.seq_timing_constraint.add ($seq_con_id.text);
//Adding timing constraints in TC_S
global_scope.seq_timing_constraint_cnt++; //N(TC_S)=N(TC_S)+1; ......................(e)
....................
```

6.3.3 Automated calculation of N(TC_T) and N(TC_{S-T}) from Timing diagram

Partial snapshot of the grammar is shown in Table 6.3.

Explanation of the Grammar

The grammar starts with the production rule timing_diagram (f) which consists of a unique sequence id (seq_id), unique timing id (time_id), one or more participants, timing parameter (time) and zero or more timing constraints(t_cons as in rule (g)).

Table 6.3: Formal grammar for the automated generation of N(TC_T) and N(TC_{S-T}) from Timing diagram

```
grammar timing;
options{ language = Java; output=AST; backtrack=true; tokenVocab = seq_dia;}
scope global{
static ArrayList t_timing_constraint; //TC_T in rule (2)
static int s_timing_constraint_cnt; //N(TC_S-T)
static int t_timing_constraint_cnt;} //N(TC_T)
import seq_dia;
timing_diagram: seq_id(time_id{
global_scope.t_timing_constraint= new ArrayList();
}) participant+ time t_cons*;............(f)
...........
t_cons:(t_cons_id { //Semantic actions
if(global_scope.seq_timing_constraint.contains ($t_cons_id.text))
//Checking existence of current token in TC_S
global_scope.s_timing_constraint_cnt++; //N(TC_S-T)=N(TC_S-T)+1
if(!global_scope.t_timing_constraint.contains
($t_cons_id.text)){ //Checking uniqueness in TC_T
global_scope.t_timing_constraint.add($t_cons_id.text);
//Adding timing constraints in TC_T
global_scope.t_timing_constraint_cnt++; //N(TC_T)=N(TC_T) +1
}) cons_type cons_desc;............(g)
.............
```

Sequence id refers to the Sequence diagram from which the Timing diagram is realized. Timing id is a unique alpha-numeric which refers to this Timing diagram.

The non-terminal t_cons_id is responsible for tracing the flow of timing constraints from a Sequence diagram to the respective Timing diagram. Each time this rule gets executed, an automated checking is performed against the timing constraints already present in the Sequence diagram. Subsequently, the count of timing constraint s_timing_constraint_cnt gets incremented by one. This counter keeps track of the number of timing constraints present in both the Sequence diagram and in the corresponding Timing diagram.

If the final value of N(TC_S) is equal to the value of N(TC_{S-T}) we can infer that all the timing constraints in the Sequence diagram have been traced in the corresponding Timing diagram.

6.3.4 Automated calculation of N(TC_{T-C}) from Class diagram

Partial snapshot of the grammar is shown in Table 6.4.

TABLE 6.4: Formal grammar for the automated generation of N(TC_{T-C}) from Class diagram

```
grammar class_dia;
scope global{
options{ language = Java; output=AST; backtrack=true; tokenVocab = timing;}
static int time_timing_constraint_cnt; } //N(TC_{T-C})
import timing;
class_diagram : classes+ relation*;
classes : t_par_id cname method_class* attribute* c_timing_cons *............(h);
.............
c_timing_cons: (c_con_id
{ //Semantic actions
if(global_scope.t_timing_constraint.contains($c_con_id.text))
/*Checking for existence of current token in (TC_{T-C})*/
global_scope.time_timing_constraint_cnt++;
//N(TC_{T-C})=N(TC_{T-C})+1............(i)
.............
```

The grammar automatically calculates the traces of timing constraints from the Timing diagram to the corresponding Class diagram using the global counter time_timing_constraint_cnt following the same process described in the previous section.

6.4 Case Study

We have considered the same case study (Patient Ventilation System) as mentioned in Chapter 4. The ventilator alarms if something untoward occurs during the surgical session. Alarming for most hazards is an appropriate safety measure for a ventilator. For safety hazards with short fault tolerance times, automatic intervention of the machine must be performed. For example, over-inflation of the lungs is a serious hazard with a fault tolerance time of about 250 ms. In this case, the ventilator will not rely on the user to correct a fault but instead will provide a secondary pressure relief valve (done mechanically) to protect the patient's lungs.

A specific form of temporal control is time determinism. Time determinism defines that an external observer can predict the points in time, at which an application produces output (can also be called time/-value determinism). This means that the observer can treat the system as a black box and predict correct values without knowing the application's internals. The points in time specifying the output times can be given by a time range or by exactly one point in time. The smaller the range, the more precise the temporal control must be. For example, in the patient ventilation system, it is easier to predict the time range of the ventilation impulse to be [0, 5) than to predict a single number such as the impulse will occur exactly every 4.7s $\pm$ 0ns.

Alarms are classified into three groups which are represented in Table 6.5:

- Informational alarms present no risk to the patient
- Caution/ warning alarms show risk (severe injury or death) within several minutes if no action is taken
- Critical alarms indicate immediate risk if no corrective action is taken

Table 6.5: Different types of alarms

Name	**Priority**	**Displayed Color**	**Description**
Informational alarm	lowest	green	displayed for a period no more than two minutes
Caution alarm	medium	yellow	displayed until acknowledged by the user within 30 sec
Critical alarm	highest	Red	re-announced after being dismissed out if the originating condition still exists

6.4.1 Sequence diagram for use case Alarm on Critical Events

• Figure 6.1 represents the sequence diagram corresponding to the use case Alarm on Critical Events of Use case diagram shown in Figure 4.7.

• The following timing constraints are represented using MARTE annotation.

o In Informational Alarm, duration between messages "Display_message_Green" and "Remove_message" must be d1 units (duration constraint).

o In Caution Alarm, the message "Ack" must be sent within d2 units (deadline constraint).

o Re-announcing Critical alarms after being dismissed out if the originating condition still exists (delay constraint).

6.4.2 Timing diagram for use case Alarm on Critical Events

• The timing diagram (corresponding to the sequence diagram (Figure 6.1)) can be represented by using MARTE annotations (Figure 6.2).

The following different timing constraints are present in the Timing diagram.

o Duration between messages "Display_Message" and "Remove_Message" must be d1 units (duration constraint).

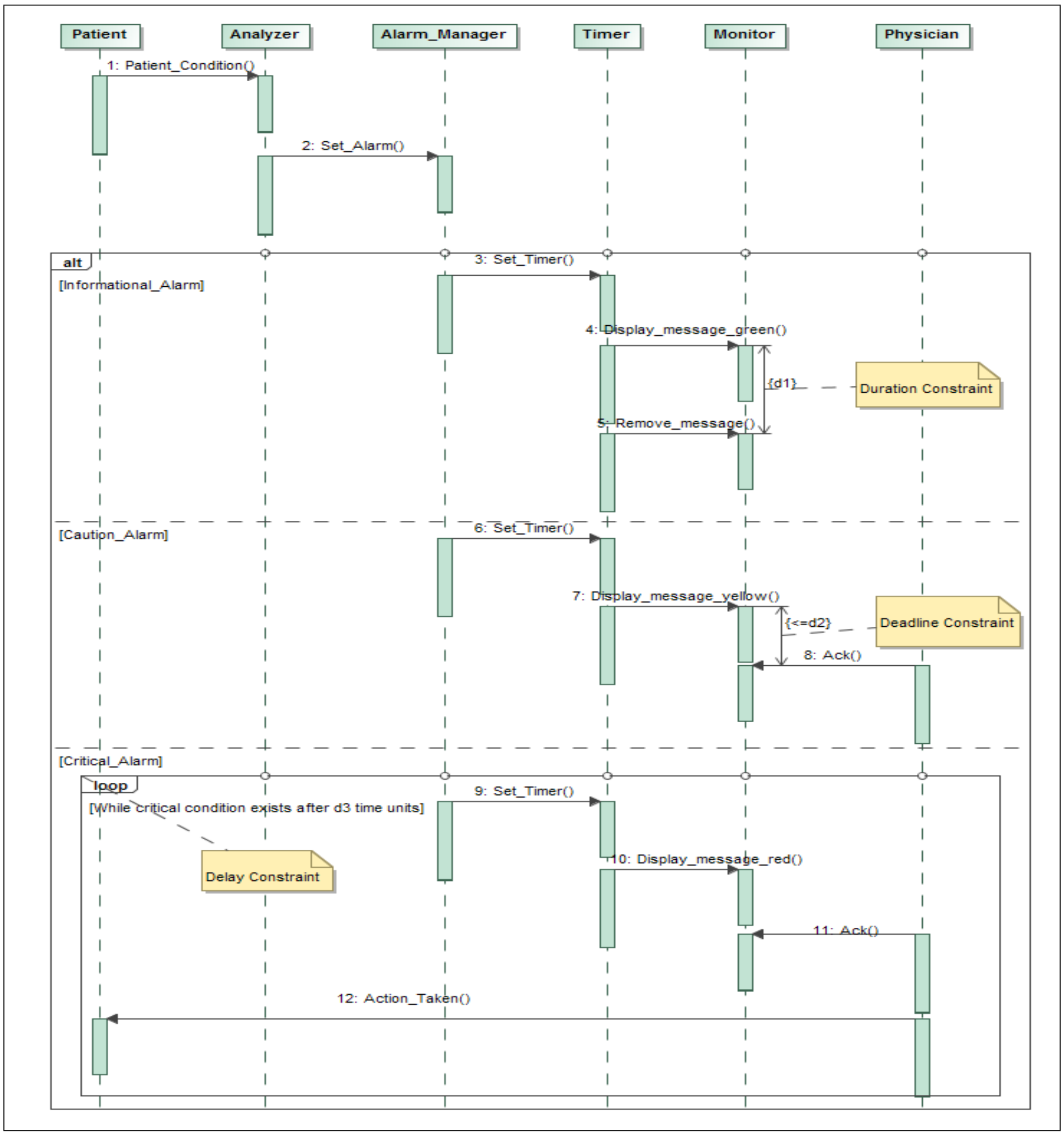


FIGURE 6.1: Sequence diagram for use case Alarm on Critical Events

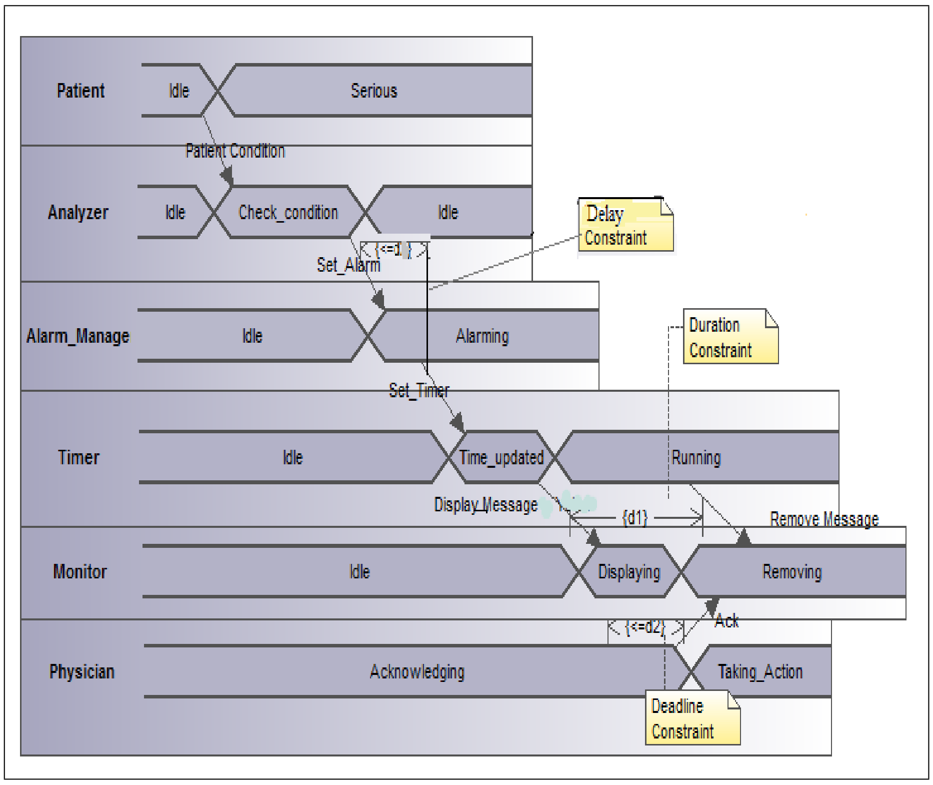


Figure 6.2: Timing diagram for use case Alarm on Critical Events

o Message "Ack" must be sent within d2 units (deadline constraint).

o Critical alarm reannounces after d3 time units if the critical condition exists (delay constraint).

6.4.3 Class diagram for use case Alarm on Critical Events

- Figure 6.3 represents the class diagram (corresponding to the timing diagram (Figure 6.2))

- The following different timing constraints are present in the Class diagram.

o Display message for d1 units (duration constraint).

o Message "Ack" must be sent within d2 units (deadline constraint).

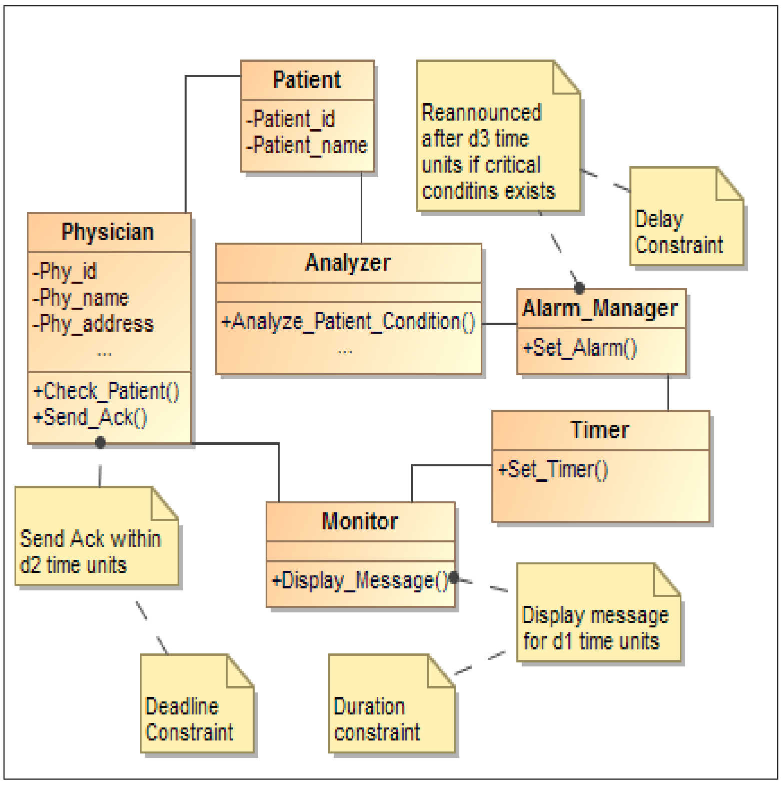


FIGURE 6.3: Class diagram for use case Alarm on Critical Events

o Critical alarm reannounces after d3 time units if the critical condition exists (delay constraint).

6.5 Automated generation of traceability metrics

Figure 6.4 represents the block diagram of the working methodology.

Most of the modeling tools (Rational Rose, Magic draw, Altova etc.) provide the facilities to generate XMI files from UML diagrams. The developed application performs the following tasks:

i) Parses the XMI file and builds input strings respectively for the grammars developed in sections 6.3.1, 6.3.2, 6.3.3, 6.3.4

ii) Executes the formal grammars and also the code embedded within the methods

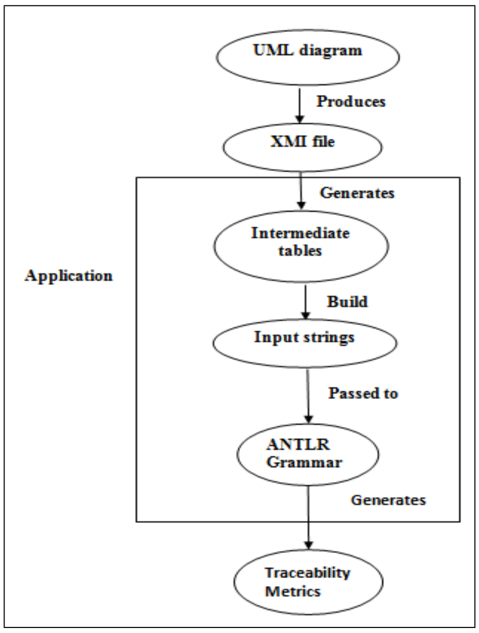


FIGURE 6.4: Block diagram of the automated generation of traceability metrics

of the formal grammars

iii) Produces traceability metrics

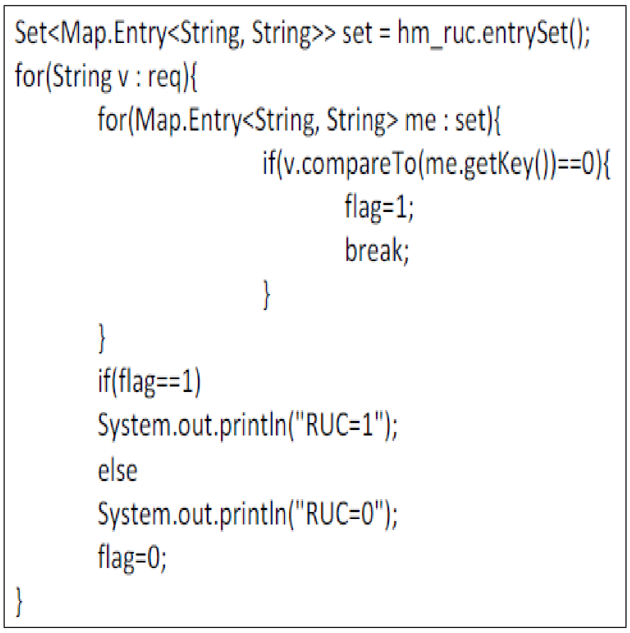
6.5.1 Results and discussion

This section represents the automated traceability measurement between the mentioned UML diagrams (Figure 4.7, Figure 6.1, Figure 6.2 and Figure 6.3) based on the developed metrics.

6.5.2 Calculation of RUC-RT

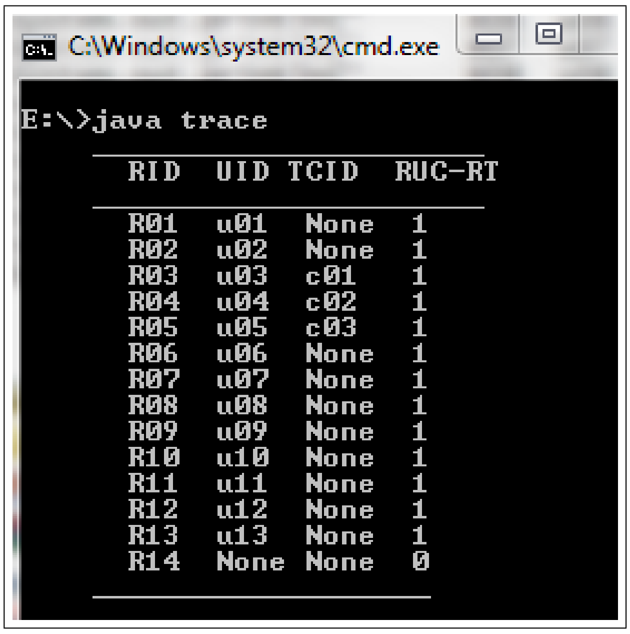
Our application automatically develops the following valid input string for the grammar presented in section 6.3.1. As use case u02 includes u03, u04 and u05; the timing constraints of these three use cases may be considered as part of u02.

R01 u01 Deliver_Ventilation R02 u02 Alarm_on_Critical_Events c02 duration descr c01 deadline descr c03 delay desc R03 u03 Informational_Alarm



```
Set<Map.Entry<String, String>> set = hm_ruc.entrySet();
for(String v : req){
        for(Map.Entry<String, String> me : set){
                        if(v.compareTo(me.getKey())==0){
                                flag=1;
                                break;
                        }
        }
        if(flag==1)
        System.out.println("RUC=1");
        else
        System.out.println("RUC=0");
        flag=0;
}
```

Figure 6.5: Code snippet for RUC Calculation



```
C:\Windows\system32\cmd.exe

E:\>java trace
          ______________________________
          RID   UID  TCID   RUC-RT
          ______________________________
          R01   u01  None    1
          R02   u02  None    1
          R03   u03  c01     1
          R04   u04  c02     1
          R05   u05  c03     1
          R06   u06  None    1
          R07   u07  None    1
          R08   u08  None    1
          R09   u09  None    1
          R10   u10  None    1
          R11   u11  None    1
          R12   u12  None    1
          R13   u13  None    1
          R14   None None    0
          __________________________
```

Figure 6.6: Output of simulator for RUC-RT Calculation

R04 u04 Caution_Alarm R05 u05 Critical_Alarm R06 u06 Set_Ventilation_Parameter u07 Monitor_Breathing_Circuit R08 u08 Set_Inspiratory_Flow R09 u09 Set_Tidal_Volume R10 u10 Set_Inspiratory_Pause R11 u11 Set_I:E_Ratio R12 u12 Set_Inspiratory_Time R13 u13 Set_Respiration_Rate R14 u14 Set_maximum_pressure_limit a01 Patient a02 Physician.

After the input string is given to the ANTLR, the system will automatically store the data values in different data structures.

ArrayList timing_constraint (TC_U) contains c01, c02 and c03 corresponding to use case u02 and static variable timing_cons_cnt (N(TC_U)) contains value 3.
Hashmap hm_ruc stores the following key value pairs:
<R01,u01>, <R02,u02>, <R03,u03>, <R04,u04>, <R05,u05>, <R06,u06>, <R07,u07>, <R08,u08>, <R09,u09>, <R10,u10>, <R11,u11>, <R12,u12>, <R13,u13> and <R14,None>. Requirement R14 is not implemented in the use case diagram.

Figure 6.5 shows the code snippet of the method use_case (generated from rule b of the grammar in section 6.3.1) for the calculation of RUC-RT. Based on the study, the requirements document listed in Table 4.1, which is generated manually. ArrayList req maintains all the requirement ids from that table. Each element of req is compared with hashmap hm_ruc for the calculation of RUC-RT. This methodology automatically generates the output as shown in Figure 6.6, which shows the mapping of requirement id to use case id, timing constraints and calculation of RUC-RT.

6.5.3 Calculation of USC-RT

We consider the use case u02 for the discussion of our work. The application creates the following input string for the grammar in section 6.3.2.

u02 s01 p01 Physician e01 event send 1 ms01 Set_Timer e01 event recv 2 c02 duration (3 - 2 $\leq$ 3) c01 deadline (2 - 1 $\leq$ 0) c03 delay (2 - 1 $\leq$ 0).

s01 is the sequence id corresponding to u02.

After the input string is given to ANTLR,
ArrayList seq_timing_constraint (TC_S) contains c01, c02 and c03. Variables use_timing_constraint_cnt (N(TC_{U-S})) and seq_timing_constraint_cnt (N(TC_S)) both contain the value 3.

Applying rule (7), we get USC-RT=3/3=1.

This methodology automatically generates Table 6.6 which represents the traceability measurement of use case into sequence diagram.

6.5.4 Calculation of STC-RT

The application automatically creates the following input string for the grammar in section 6.3.3.

s01 t01 p01 physician st01 Acknowledging e01 event ms01 Ack p01 st01 Acknowledging st02 removing 1 c02 duration desc1 c01 deadline desc2.

After the input string is given to the ANTLR,
ArrayList t_timing_constraint (TC_T) contains c01 and c02. Variables t_timing_constraint_cnt (N(TC_T)) and s_timing_constraint_cnt (N(TC_{S-T})) contain the value 3 and 2 (i.e., missing one timing constraint) respectively.
Applying rule (8), we get STC-RT= 2/3=0.67.

Table 6.7 is developed automatically from the framework which helps to detect missing requirements (R05) easily.

6.5.5 Calculation of TCC-RT

Application automatically creates the following input string for the grammar in section 6.3.4.

Table 6.6: Calculation of USC-RT

RID	**UID**	**N(TC_U)**	**N(TC_{U-S})**	**USC-RT**
R02	u02	3	3	1
R03	u03	1	1	1
R04	u04	1	1	1
R05	u05	1	1	1

Table 6.7: Calculation of STC-RT

RID	**N(TC_S)**	**N(TC_{S-T})**	**STC-RT**
R02	3	2	0.67
R03	1	1	1
R04	1	1	1
R05	0	0	0

Table 6.8: Calculation of TCC-RT

RID	**N(TC_T)**	**N(TC_{T-C})**	**TCC-RT**
R02	2	2	1
R03	1	1	1
R04	1	1	1

p01 Physician M01 + void Check_Patient () M02 + void Send_Ack () - int Phy_id - String Phy_name - String Phy_address c01 duration desc1 p02 Monitor M03 + void Display_messager( ) c02 deadline desc2 c03 delay des3 Monitor 1..2 physician id descr3 association.

After the input string is given to the ANTLR, the system will automatically store the data values like static variable time_timing_constraint_cnt (N(TC_{T-C})) which contains value 2. N((TC_T)) value is available from the grammar of timing grammar. Applying rule (9), TCC-RT= 2/2 =1

Similarly, our framework ensures the automated generation of TCC-RT (Table 6.8). Finally, from Figure 6.6, Table 6.6, Table 6.7 and Table 6.8 we measure the RCF-RT value for requirements R02, R03, R04 and R05 as shown in Table 6.9.

RCM-RT = (0.25+0.92+1+1+0.5+0.25+0.25+0.25+0.25+0.25+0.25+0.25+0.25)/14 = 0.40

Table 6.9: Calculation of RCF-RT

RID	RUC-RT	USC-RT	STC-RT	TCC-RT	RCF-RT
R01	1	0	0	0	0.25
R02	1	1	0.67	1	0.92
R03	1	1	1	1	1
R04	1	1	1	1	1
R05	1	1	0	0	0.5
R06	1	0	0	0	0.25
R07	1	0	0	0	0.25
R08	1	0	0	0	0.25
R09	1	0	0	0	0.25
R10	1	0	0	0	0.25
R11	1	0	0	0	0.25
R12	1	0	0	0	0.25
R13	1	0	0	0	0.25
R14	0	0	0	0	0

This means that 40% of the requirements have been implemented in the design phase. Our framework ensures automated generation of traceability metrics as well as detection of missing requirements at the early stage of design.

6.6 Comparison with related work

Authors in [105] combine the application of SysML with MARTE stereotypes to specify the important elements of individual software requirements for RTSS. In any case, the work does not verify the traceability between requirement and design phase of the software development life cycle and furthermore does not provide any formal specification to check the non-functional requirements in the outline. The fundamental commitment of [58] is to develop a set of metrics based on use case, sequence and class diagrams. However, authors have worked on non-real-time systems, and hence scope excludes any real-time requirements such as timing constraints. In comparison to those research works, this Chapter further enhances the concept of RTSS and shows how requirements related to timing constraints are traced automatically from the use case diagram through the sequence diagram to the timing diagram and finally to the class diagram.

6.7 Threats to validity of the proposed approach

We have developed a simulator tool that automatically generates the traceability metrics for timing constraints from a given set of requirements and UML design models. Most of the modeling tools (Rational Rose, Magic draw, Altova etc.) provide the facilities to generate XMI files from UML diagrams. Different modeling tools generate different formats of XMI files. In this research work, we have used XMI file generate from Magic draw UML. Our simulator tool parses this specific XMI file and builds input strings respectively for the proposed grammars. This simulator tool may need to be modified to handle the XMI file developed from other modelling tools.

6.8 Conclusion

UML is frequently applied to contribute to deal with the multifaceted nature of RTSS advancement. The ever-increasing design complexity of that system is always squeezing the interest for more abstract design levels and conceivable techniques for formal verification which has become important to ensure the development of more reliable systems. This chapter presents a comprehensive framework for ensuring traceability of timing constraints from the requirements analysis phase into the design phase. This framework automatically generates the trace metrics based on UML diagrams, demonstrates the degree of coverage of timing requirements and finally detects any missing requirements.

Chapter 7

Schedulability Analysis of Real-Time Software Systems

7.1 Introduction

The real-world is inherently concurrent, and a real-time system which is linked to the behavior of the real world must behave in a concurrent manner. Because of this concurrency, there may be contention for resources, requiring scheduling (i.e. how tasks are granted access to a given resource). In real-time systems, the scheduling of tasks with hard deadlines has been an important area of research.

The main contribution of this chapter is to model, analyze and verify the existing resource access control protocols (Priority Inheritance, Priority Ceiling, Stack Based Priority Ceiling and Stack Based Preemption Ceiling Protocols) using sequence and timing diagrams. Table 7.1 shows a comparison of the above mentioned protocols. From the comparison, it is observed that the Stack Based Preemption Ceiling Protocol (SBPCP) can be more appropriate to schedule the dynamic tasks in a shared resource environment.

7.2 Real-Time Scheduling

A real-time system is one in which failure can occur in the time domain as well as in the more familiar value domain. The correctness of the system depends on the completion time of the tasks. Producing a response later than the completion time, which is also known as deadline missing in real-time systems, will be erroneous and may have a significant impact to the system.

A scheduler is one which provides an algorithm or policy for ordering the execution of the tasks on the processor according to some predefined criteria. Schedulers may be preemptive or non-preemptive. The former can arbitrarily suspend a process's execution and restart it later without affecting the behavior of that process (except by increasing its elapsed time). Preemption typically occurs when a higher priority process becomes runnable. The effect of preemption is that a process may be suspended involuntarily. A non-preemptive scheduler does not suspend a process in this way. When a task holds any resource, it executes at a priority higher than the priorities of all other tasks. The preemptive scheduler can arbitrarily suspend a process's execution and restart it later without affecting the behavior of that process (except by increasing its elapsed time). Preemption typically occurs when a higher priority process becomes runnable. The effect of preemption is that a process may be suspended involuntarily.

Real-Time Systems need to share resources among tasks. Serially reusable resources are typically allocated to tasks on a non-preemptive basis and used in a mutually exclusive manner. In other words, when a unit of a resource Ri is granted to a job, this unit is no longer available to other jobs until the job frees the unit.

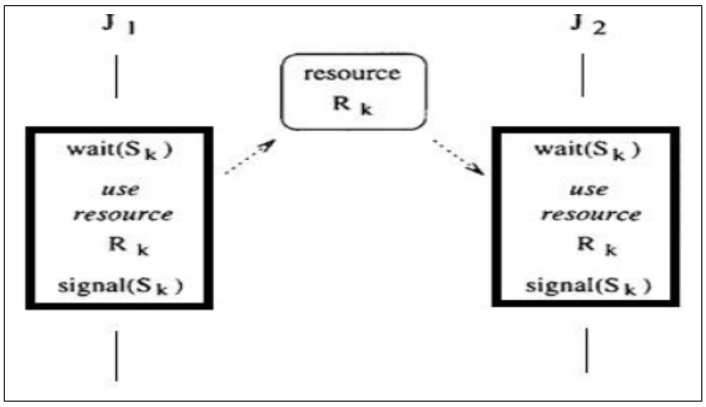


FIGURE 7.1: Structure of two tasks that share an exclusive resource

7.2.1 Priority Inversion

Priority inversion occurs when the execution of some jobs or portions of jobs is non-preemptable. Resource contentions among jobs can also cause priority inversion. Because resources are allocated to jobs on a non-preemptive basis, the highest priority active task cannot execute because some of the resources needed for its execution are held by some other tasks. At that point in time, the higher priority task is blocked while the lower-priority tasks execute.

Considering two tasks J1 (higher priority) and J2 (lower priority) that share an exclusive resource Rk, on which two operations (such as insert and remove) are defined. To guarantee mutual exclusion, both operations must be defined as critical sections. If a binary semaphore Sk is used for this purpose, then each critical section must begin with a wait(Sk) primitive and must end with a signal(Sk) primitive (Figure 7.1). When J2 is using Rk, at that time J1 requires Rk. So, J1 has to wait until J2 completes its execution with Rk. As a result, priority inversion occurs in this situation.

In order to overcome the problems of priority inversion, the Priority Inheritance Protocol can be used.

7.2.2 Priority Inheritance Protocol

Assigned Priority: When a task releases, this priority is assigned to the task. It is a unique priority.

Current Priority: It is the priority at which a ready task is scheduled and executed. It may vary with time.

Rules of the PIP

1. Scheduling rule: A ready task is scheduled preemptively in a priority-driven manner according to the current priority.
2. Allocation rule: When a task T requests a resource R,
   (a) If R is free it is allocated to T and is held by T until T releases R.
   (b) If R is not free then the task is blocked.
3. Priority Inheritance rule: When the requesting task T becomes blocked, the task Tl which blocks T inherits the current priority π(t) of T. When Tl releases R its priority reverts to π(t'), where t' is the time it acquired the resource R.

Again from the above rules, we conclude that the Priority Inheritance Protocol has some advantages as well as disadvantages. They are as follows:

Advantages:

a) The protocol prevents non-preemptivity between two jobs in a task.

b) It also prevents priority inversion.

Disadvantages:

a) The protocol does not prevent deadlock.

b) It also creates indefinite blocking.

The Priority Ceiling Protocol removes these drawbacks.

7.2.3 Basic Priority Ceiling Protocol

This protocol makes two key assumptions:

i) The assigned priority of all tasks is fixed.

ii) The resources required by all tasks are known a priori before the execution of any task begins.

The priority ceiling of a critical resource R is the highest priority of all the tasks that use the resource R. Current priority ceiling of a system π'(t) at any time t is equal to the highest priority ceiling of all the resources used at that time. If all the resources are free, then π'(t)=Ω where Ω is a non-existing priority level that is lower than the lowest priority of all tasks.

Rules of the Basic PCP

1. Scheduling rule:- At the release time t, the current priority π(t) of every task is equal to the assigned priority. The task remains at that priority level except by rule 3.
2. Allocation rule:- When a task T requests R, one of the following conditions occur:
    (a) If R is not free then T becomes blocked.
    (b) If R is free then one of the following conditions occur:
        i. T's priority is higher than the current priority ceiling π'(t), R is allocated to T.
        ii. If T's priority is not higher than the priority ceiling π'(t), R is allocated to T only if T is the task holding the resource whose priority ceiling is π'(t). Otherwise, T's request is denied.

3. Priority Inheritance rule:- When T becomes blocked, the task Tl which blocks T inherits the current priority π(t) of T. Tl executes at its inherited priority until the time it releases every resource whose priority ceiling is equal to higher than π(t). At that time priority of Tl returns to its priority π'(t') at the time t' when it was granted the resource.

7.2.4 Stack Based Priority Ceiling Protocol

A resource in the system is the run-time stack. Thus far, it has been assumed that each job has its run-time stack. Sometimes, especially in systems where the number of jobs is large, it may be necessary for the jobs to share a common run-time stack, to reduce overall memory demand. When a job J executes, its stack space is on the top of the stack.

Space is freed when the job completes. To ensure deadlock-free sharing of the run-time stack among jobs, we must ensure that no job is ever blocked because it is denied some resource once its execution begins. This observation leads to the following modified version of the priority ceiling protocol, called the stack-based priority ceiling protocol. Like Baker's protocol, this protocol allows jobs to share the runtime stack if they never self-suspend. In the statement of the rules of the stack-based priority ceiling protocol, again the term (current) ceiling π(t) of the system has been used, which is the highest priority ceiling of all the resources that are in use at time t. If all the resources are free, then π'(t)=Ω where Ω is a non-existing priority level that is lower than the lowest priority of all tasks.

This protocol makes two key assumptions:

1. The assigned priorities of all tasks are fixed.

2. The resources required by all tasks are known a priori before the execution of any task begins.

Rules of the SPCP

1. Scheduling rule: After a task is released, it is blocked from starting execution until its assigned priority is higher than the current ceiling π'(t) of the system. At all times, the tasks that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.
2. Allocation rule: Whenever a task requests a resource, it is allocated to the task.

More importantly, no job is ever blocked once its execution begins. Likewise, when a job J is preempted, all the resources the preempting job will require are free, ensuring that the preempting job can always complete so J can resume. Consequently, deadlock can never occur.

7.2.5 Stack Based Preemption Ceiling Protocol

The advantages of using the Stack Based Preemption Ceiling Protocol over the Dynamic Priority Ceiling Protocol are as follows:

1. It avoids paying the time or storage overhead.
2. Ceilings are defined in terms of preemption levels, instead of priorities, so that this protocol applies directly to Earliest Deadline First (EDF) scheduling (without dynamic recomputation of ceilings).
3. Potential of resource contentions in dynamic priority systems does not change with time.
4. Stack sharing is supported.
5. Resource requests never block, hence do not require extra context switches.

6. Because there is no blocking after a job starts executing, a stronger EDF schedulability result can be obtained than with dynamic priority ceilings.

This protocol can be used to prevent deadlock.

The Preemption level of a task (π(T)) will be inversely proportional to the relative deadline of the task. The reason for introducing the Preemption level is to enable us to do static analysis of potential resource conflicts, even for dynamic priority schemes such as EDF scheduling. The essential property of this protocol is that a task Ti' isn't allowed to preempt another task Ti unless π(Ti')>π(Ti.).

The preemption ceiling of a critical resource R is the highest preemption level of all the tasks that use the resource R. The (preemption) ceiling of a system π'(t) at any time 't' is equal to the highest preemption ceiling of all the resources used at that time. If all the resources are free, then π'(t)=Ω where Ω is a non-existing preemption level that is lower than the lowest preemption level of all tasks.

Rules of the SBPCP

1. Scheduling rule: After a task is released, it is blocked from starting the execution until its preemption level is higher than the current ceiling π'(t) of the system and the preemption level of the executing task. At any time t, tasks that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.

2. Allocation rule: Whenever a task requests a resource R, it is allocated to the task.

3. Priority Inheritance Rule: When some task is blocked from starting, the blocking task inherits the highest priority of all the blocked tasks.

7.3 Comparative analysis of various resource access control protocols

TABLE 7.1: Comparison among various resource access control protocols

Comparison Criteria	**PIP**	**PCP**	**SPCP**	**SBPCP**
Nature of priority	Static	Static	Static	Static and dynamic
Blocking	More	More	Less	Less
Context switching	More	More	Less	Less
Deadlock prevention	No	Yes	Yes	Yes

In RTSS, the main intent of various resource access control protocols is to schedule and synchronize different tasks when many of these use the same shared resources. Hard real-time tasks have stringent timing constraints. Resource access control protocols employ blocking to resolve data conflicts among tasks when tasks concurrently access the shared data. The concurrency control protocols adapted in priority-driven scheduling pose the priority inversion problems. Unfortunately, tasks blocking due to priority inversion can be indefinitely long. This unpredictability of task execution is unacceptable in most mission-critical applications. We have compared above mentioned resource access control protocols in Table 7.1.

7.4 Modeling of resource access control protocols

In RTSS, the scheduling of tasks with hard deadlines has been an important area of research. The main objective of this research work is to compare various protocols using UML models. The shortcomings of the existing Priority Inheritance protocol are represented using one UML/SPT model. Further, various resource access control protocols are used to overcome this difficulty using other improved models. Using UML/SPT Sequence and Timing Diagrams, we model the above mentioned protocols. Further, we analyze these to highlight deadlock occurrence and deadlock avoidance.

7.4.1 Sequence diagram and Timing diagram

In this work, both sequence and timing diagrams are used because the sequence diagram or timing diagram alone does not depict the scenario completely. The essential difference between the sequence and timing diagrams is that the latter emphasizes the change in value or state over time while the former emphasizes sequences of message exchange. They are, approximately at least, isomorphic and able to represent the same information, but their purpose is different. The Sequence diagram focuses primarily on the sequences of messages in operational scenarios. Timing diagram focuses on the qualities of service having to do with time, such as execution time, jitter, deadlines, periodicity, and so on, and how they affect the state of the system (or, more precisely, of the use case) or an important value held by the system.

7.4.2 Advantages of Sequence diagram over Timing diagram

The Sequence diagram is used to model the flow of messages, events and actions between the objects or components of a system.

The Sequence diagram is often used to design the interactions between components of a system that need to work together to accomplish a task. It focuses on when the individual objects interact with each other during execution. It is particularly useful for modeling usage scenarios such as the logic of methods and the logic of services.

The Sequence diagram emphasizes message sequence, so the time of the next message is the message following the current one on the diagram.

The Timing diagram does not represent these.

7.4.3 Advantages of Timing diagram over Sequence diagram

A Timing diagram is a simple representation with time along the horizontal axis and objects state or attribute value along the vertical axis.

Although the Timing diagram does not show any information beyond that available in the annotated Sequence diagram, the absolute timing of events, state changes and the relative timing among the lifelines is clearer and more readable than Sequence diagram, even when explicit timing constraints are added. Messages on the Sequence diagram are only partially ordered, so in many cases, the relative timing between two messages is not specified.

When messages on the Sequence diagram begin or finish on different lifelines, it is not possible to compare which one starts or terminates first.

Time goes down the page on the Sequence diagram, but usually, linearity is not implied; that is, further down implies later in time, but the same distance at different places in the diagram does not imply the same amount of time. However, each diagram provides different points of view to the same scenario and both could be very useful.

7.4.4 UML/SPT

UML profile for Schedulability, Performance and Time (UML/SPT) is a UML profile for real-time modeling. UML/SPT is a framework to model resource, time, concurrency concepts as well as the quality of services. Besides, UML/SPT also can support ordinary UML models for predicting quantitative analysis, which includes schedulability and performance analysis [45]. UML/SPT provides a set of stereotypes to the designer and the developers and tagged values to interpret the UML models.

This chapter focuses on the UML/SPT based Sequence and Timing diagrams to model real-time resource access control protocols.

7.4.4.1 UML/SPT Sequence diagram

In UML/SPT, the notation for an interaction in a sequence diagram is a solid-outline rectangle (a rectangular frame). The five-sided box at the upper left-hand corner names the sequence diagram: keyword sd followed by the interaction name. Each lifeline in the diagram represents an individual participant in the scenario.

s1: Scheduler. A scheduler (in our domain, a processor) is responsible for processing the acquisition requests from the clients of service and based on the appropriate access control policy for that service, it dispenses access to the service. If a service instance is busy, then the reply may remain pending until the access is possible. The scheduler determines a schedule that allocates a set of scheduling tasks to its set of execution engines.

r1, r2: Resources. The stereotype <<SAresource>> of the UML Profile for Schedulability, Performance and Time (schedulability modeling) represents a kind of protected resource (e.g., a semaphore) that is accessed during the execution of a scheduling task. It may be shared by multiple concurrent actions and must be protected by a locking mechanism. The tag "SAaccessControl" represents the access control policy for handling requests from scheduling tasks (in our model, 'Priority Inheritance').

T1, T2, T3: Tasks. The stereotype <<SAschedRes>> of the UML Profile for Schedulability, Performance, and Time (schedulability modeling) represents a unit of concurrent execution (in our domain, a task), which is capable of executing a single scenario concurrently with other concurrent units. In the general resource modeling of the UML Profile for Schedulability, Performance and Time, an action is defined as a kind of scenario. Therefore, the stereotype <<SAaction>> of this profile (schedulability modeling) is used to characterize the behavior of each task in the developed model. The new metaclass in UML/SPT, TimeObservationAction, is used to know when a task awakes. A time observation triggers an action that, when executed, returns the current value of time in the context in which it is executing. It is depicted with the keyword ”now”. Another new metaclass in UML/SPT, StateInvariant, is used to show the different states associated with each lifeline as restrictions. A state invariant is a constraint on the state of a lifeline. If the constraint is true, the trace is valid. Finally, notes are used to display textual information.

7.4.4.2 UML/SPT Timing diagram

The timing diagram can be stereotyped as <<SAsituation>> to use it in the context of schedulability analysis, representing a real-time situation.

The notations of the rectangular frame and the five-sided box are the same as in the previous sequence diagram, but now we have different elements in the model. Five lifelines are generated one each for the two resources (r1, r2) and the three tasks (T1, T2 and T3) respectively. In this case, scheduler (s1) can be ignored, because it is not necessary to understand the scheduling. Since the changes in states of different lifelines can be represented over linear time, there is no need to show the message passing.

The task states used in the timing diagram are explained in Table 7.3. There are two simple states for the resource lifeline: idle and busy. Using the timing diagrams it can be seen how the states get changed over time for each lifeline. Therefore, it is not required to use the metaclass StateInvariant as a restriction in lifelines to know the state value at a particular time.

The time axis is linear so it clarifies the absolute timing of events, state changes and relative timing between the different lifelines. Therefore, it is not required to use notes indicating when a task awakes (when the state of a task changes to ”Ready”) [131].

7.4.5 Case study

In [131], the Priority Inheritance Protocol is used for sharing critical resources but this protocol does not prevent deadlock. In this research work, the occurrence of deadlock is first highlighted by considering the following task set which is described by the classical parameters given in Table 7.2. Further, deadlock avoidance is discussed using the Basic Priority Ceiling Protocol, Stack Based Priority Ceiling Protocol and Stack Based Preemption Ceiling Protocol by considering the same task set.

System description

Consider a set T of n simple independent tasks $\{T_1, T_2,...,T_n\}$ to be scheduled preemptively on uniprocessor system. Each task T_i (1≤i≤n) consists of a potentially unbounded stream of jobs and is characterized by release time Re_i, priority P_i, relative deadline D_i, execution time E_i, preemption level PL_i and its resource requirements (discussed in Table 7.2).

Three parameters Ea, Eb and Ec related to the execution time are defined as follows:

-Ea: Execution time before holding the resource

-Eb: Execution time using the resource

-Ec: Execution time after releasing the resource

Then, the execution time E=Ea+Eb+Ec

The system also comprises of m non-preemptable serially reusable resources {R1, R2, . . . , Rm}.

Table 7.2: Task set sharing critical resources

$Task$	Re_i	E_i	E_a	E_b	E_c	P_i	D_i	PL_i
T_1	4	2	1	(R1, 1)	0	1	7	1
T_2	2	5	1	(R2, 3)(R1, 1)	0	2	9	2
T_3	0	5	1	(R1, 3)(R2, 1)	0	3	12	3

7.5 Modeling and analysis of PIP to highlight deadlock occurrence

Priority Inheritance Protocol can be used to overcome the priority inversion problem, but it does not prevent deadlock. Deadlock occurrence is modeled using sequence diagram(Figure 7.2) and timing diagram (Figure 7.3).

7.5.1 Observation

The sequence diagram in Figure 7.2 shows that deadlock occurs. T1 is blocked by T3. T3 is waiting for a resource that is held by T2. T2 is waiting for a resource that is held by T3. As a result, all of the three tasks are blocking.

The timing diagram in Figure 7.3 describes how deadlock occurs in the Priority Inheritance Protocol.

7.5.2 Result and discussion

At time 0, task T3 is released and executes at its assigned priority 3. At time 1, resource R1 is assigned to T3.

At time 2, T2 is released. It preempts T3 (as priority of T2 is greater than the priority of T3) and starts executing.

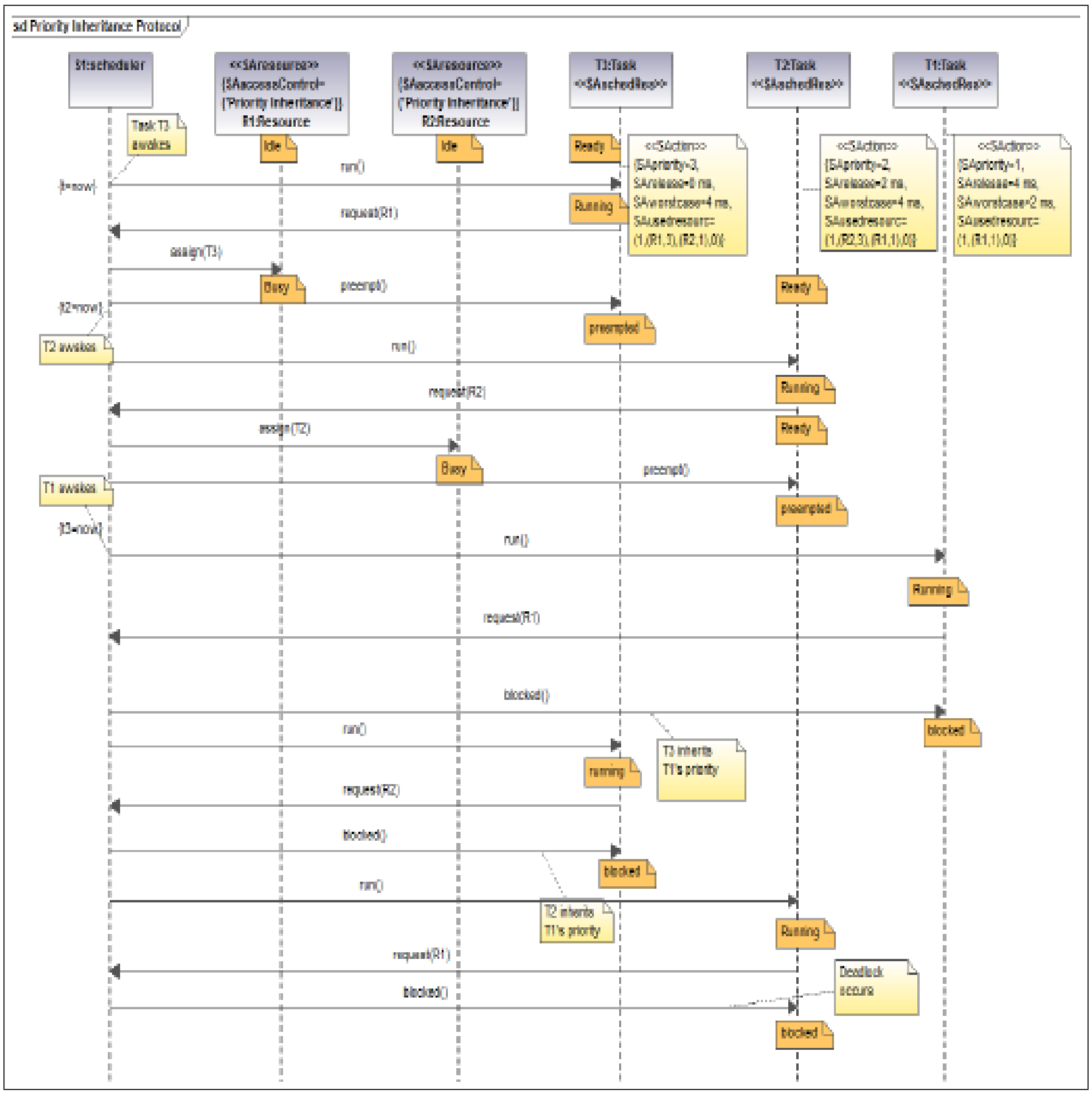

sd Priority Inheritance Protocol
S1:scheduler
<<SAresource>> {SAaccessControl='Priority Inheritance'} R1:Resource
<<SAresource>> {SAaccessControl='Priority Inheritance'} R2:Resource
T3:Task <<SAschedRes>>
T2:Task <<SAschedRes>>
T1:Task <<SAschedRes>>
Task T3 awakes
Idle
Idle
Ready
<<SAction>> {SApriority=3, SArelease=0 ms, SAworstcase=4 ms, SAusedresource= (1,(R1,3),(R2,1),0)}
<<SAction>> {SApriority=2, SArelease=2 ms, SAworstcase=4 ms, SAusedresource= (1,(R2,3),(R1,1),0)}
<<SAction>> {SApriority=1, SArelease=4 ms, SAworstcase=2 ms, SAusedresource= (1,(R1,1),0)}
{t=now}
run()
Running
request(R1)
assign(T3)
Busy
preempt()
Ready
{t2=now}
preempted
T2 awakes
run()
Running
request(R2)
Ready
assign(T2)
Busy
preempt()
T1 awakes
preempted
{t3=now}
run()
Running
request(R1)
blocked()
blocked
run()
T3 inherits T1's priority
running
request(R2)
blocked()
blocked
run()
T2 inherits T1's priority
Running
request(R1)
Deadlock occurs
blocked()
blocked

FIGURE 7.2: Sequence diagram showing deadlock occurrence using PIP

At time 3, T2 requests resource R2. R2, being free, is assigned to T2. The task T2 continues to execute. At time 4, T1 is released and it preempts T2 (as priority of T1 is greater than the priority of T2).

At time 5, T1 requests R1 but R1 is already assigned to T3. So T1 is now directly blocked by T3 though the priority of T1 is greater than the priority of T3. According to Rule 3 in section 7.2.2, T3 inherits T1's priority (i.e. 1) and T3 continues execution.

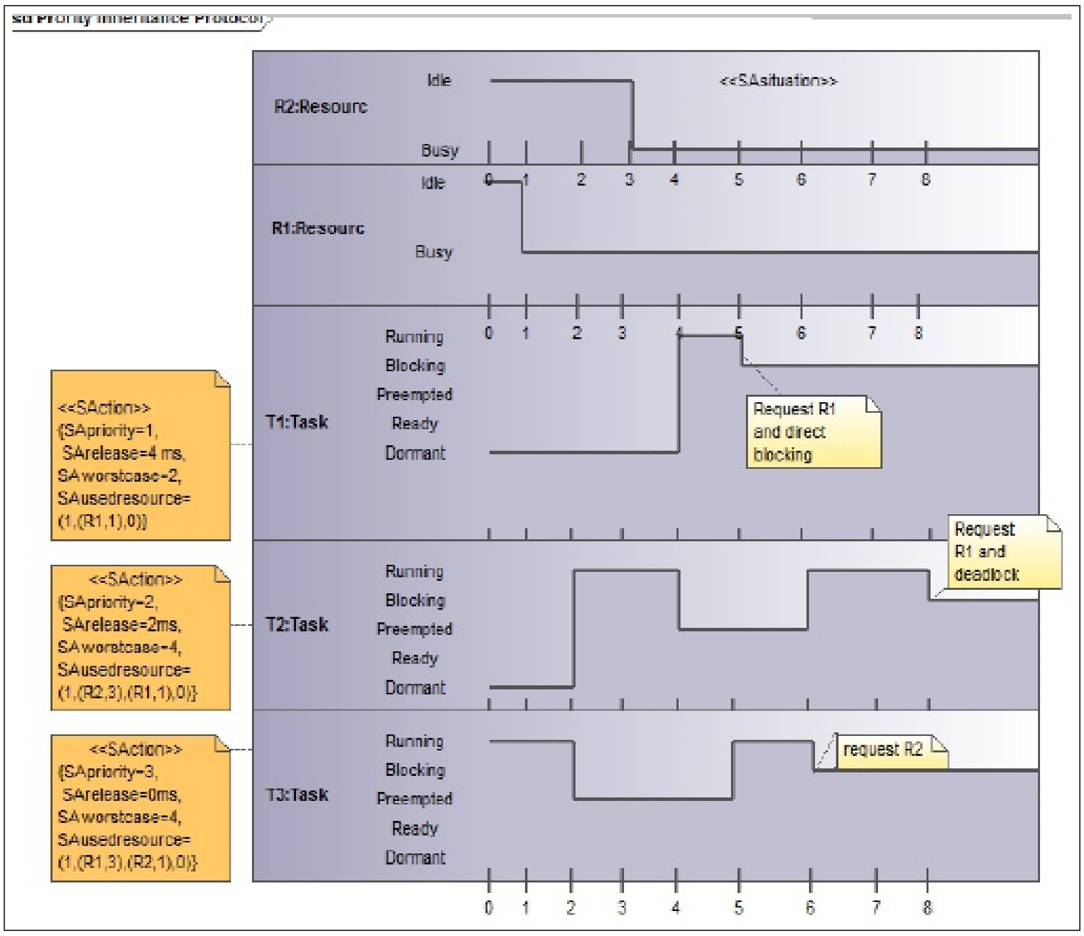

sd Priority Inheritance Protocol
R2:Resourc
Idle
Busy
<<SAsituation>>
R1:Resourc
T1:Task
T2:Task
T3:Task
Running
Blocking
Preempted
Ready
Dormant
0 1 2 3 4 5 6 7 8
<<SAction>> {SApriority=1, SArelease=4 ms, SAworstcase=2, SAusedresource=(1,(R1,1),0)}
<<SAction>> {SApriority=2, SArelease=2ms, SAworstcase=4, SAusedresource=(1,(R2,3),(R1,1),0)}
<<SAction>> {SApriority=3, SArelease=0ms, SAworstcase=4, SAusedresource=(1,(R1,3),(R2,1),0)}
Request R1 and direct blocking
Request R1 and deadlock
request R2

FIGURE 7.3: Timing diagram showing deadlock occurrence using PIP

At time 6, T3 requests R2 but R2 is already assigned to T2. So T3 is blocked by T2 though the current priority of T3 (presently the priority of T3 is 1 which it inherits from T1) is greater than the priority of T2. According to the same Rule 3, T2 inherits T3's priority (i.e. 1) and T2 continues execution.

At time 8, T2 requests R1 but R1 is already assigned to T3, so T2 is blocked by T3. As T3 is already blocked by T2, a deadlock occurs.

Table 7.3: Task states

State	Description
Dormant	The task is set up
Ready	The task awakes
Preempted	When running, the task is preempted
Blocked	The task is waiting for a signal or a resource
Running	Assignment of processor to task

7.6 Modeling and analysis of resource access control protocols to prevent deadlock

Basic PCP, SPCP and SBPCP are used to prevent the deadlock.

7.6.1 Basic PCP

7.6.1.1 Observation

The sequence diagram (Figure 7.4) shows that deadlock does not occur for this protocol. All the three tasks complete their executions successfully.

The Timing diagram (Figure 7.5) shows how deadlock is prevented.

7.6.1.2 Result and discussion

T3 is released at time 0. The ceiling of the system at time 1 is Ω. When T3 requests R1, it is allocated to T3 according to (i) in part (b) of Rule 2 of section 7.2.3. After the allocation of R1, the ceiling of the system is raised to 1, the priority ceiling of R1.

At time 2, T2 is released and it preempts T3 (as the priority of T2 is greater than the priority of T3). At time 3, T2 requests resource R2. R2 is free; however because the ceiling π'(3)(=1) of the system is higher than the priority of T2, T2's request is

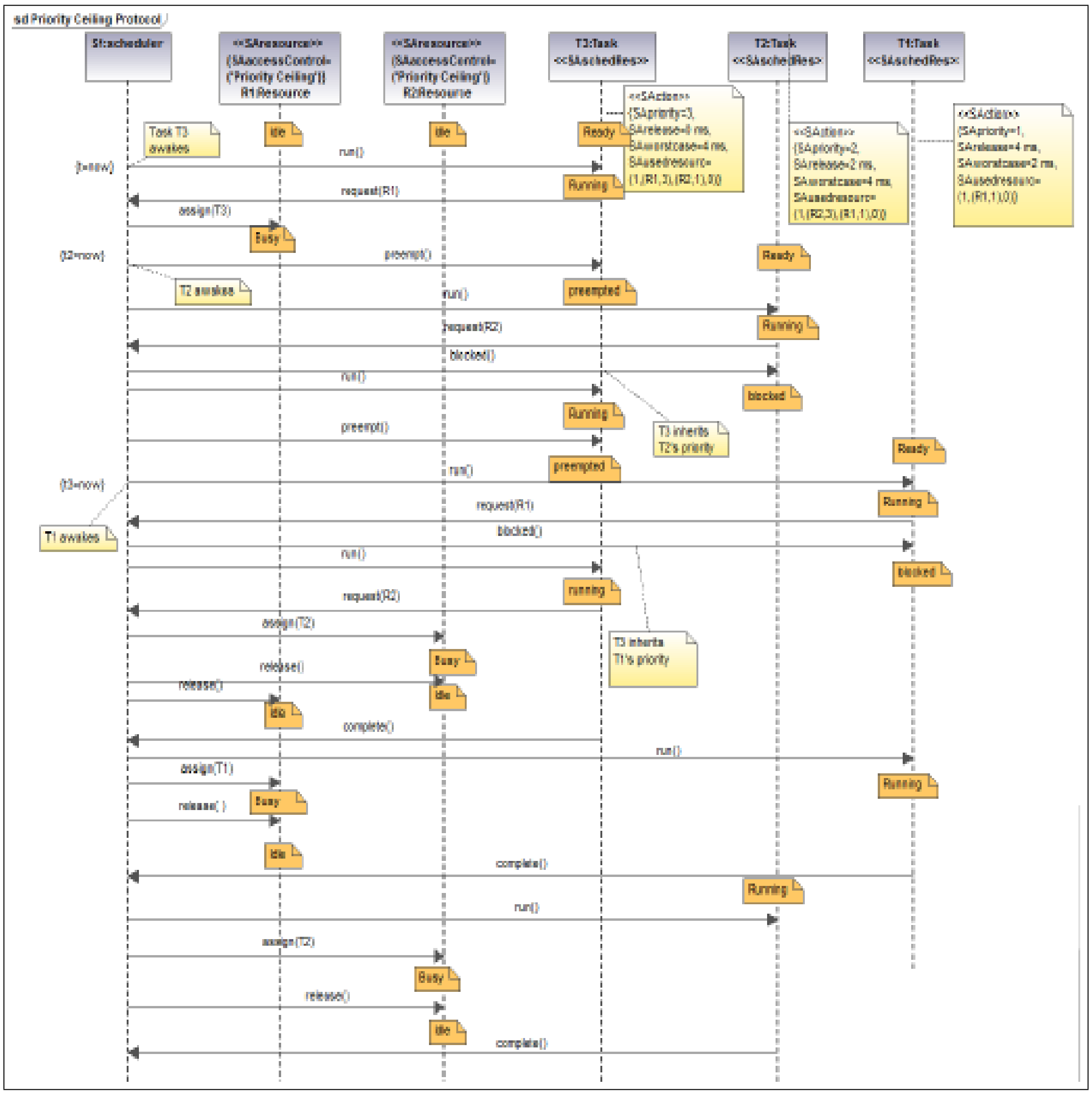

sd Priority Ceiling Protocol
S1:scheduler
<<SAresource>> {SAaccessControl=('Priority Ceiling')} R1Resource
<<SAresource>> {SAaccessControl=('Priority Ceiling')} R2Resource
T3:Task <<SAschedRes>>
T2:Task <<SAschedRes>>
T1:Task <<SAschedRes>>
<<SAction>> {SApriority=3, SArelease=0 ms, SAworstcase=4 ms, SAusedresource= (1,(R1,3),(R2,1),0)}
<<SAction>> {SApriority=2, SArelease=2 ms, SAworstcase=4 ms, SAusedresource= (1,(R2,3),(R1,1),0)}
<<SAction>> {SApriority=1, SArelease=4 ms, SAworstcase=2 ms, SAusedresource= (1,(R1,1),0)}
Task T3 awakes
{t=now}
{t2=now}
{t3=now}
T2 awakes
T1 awakes
T3 inherits T2's priority
T3 inherits T1's priority
Idle
Ready
Running
Busy
preempted
blocked
running
run()
request(R1)
request(R2)
assign(T3)
assign(T2)
assign(T1)
preempt()
blocked()
release()
complete()

FIGURE 7.4: Sequence diagram showing deadlock prevention using PCP

denied according to (ii) in part (b) of Rule 2 of section 7.2.3. T2 is blocked and T3 inherits T2's priority.

At time 4, T1 is released and it preempts T3 (as priority of T1 is greater than the priority of T3). At time 5, T1 requests resource R1 and becomes directly blocked by T3 and T3 inherits T1's priority. At time 5, T3 requests for resource R2. R2 is free and it is allocated to T3 because T3 is holding the resource R1 whose priority ceiling is equal to π'(t)(=1).

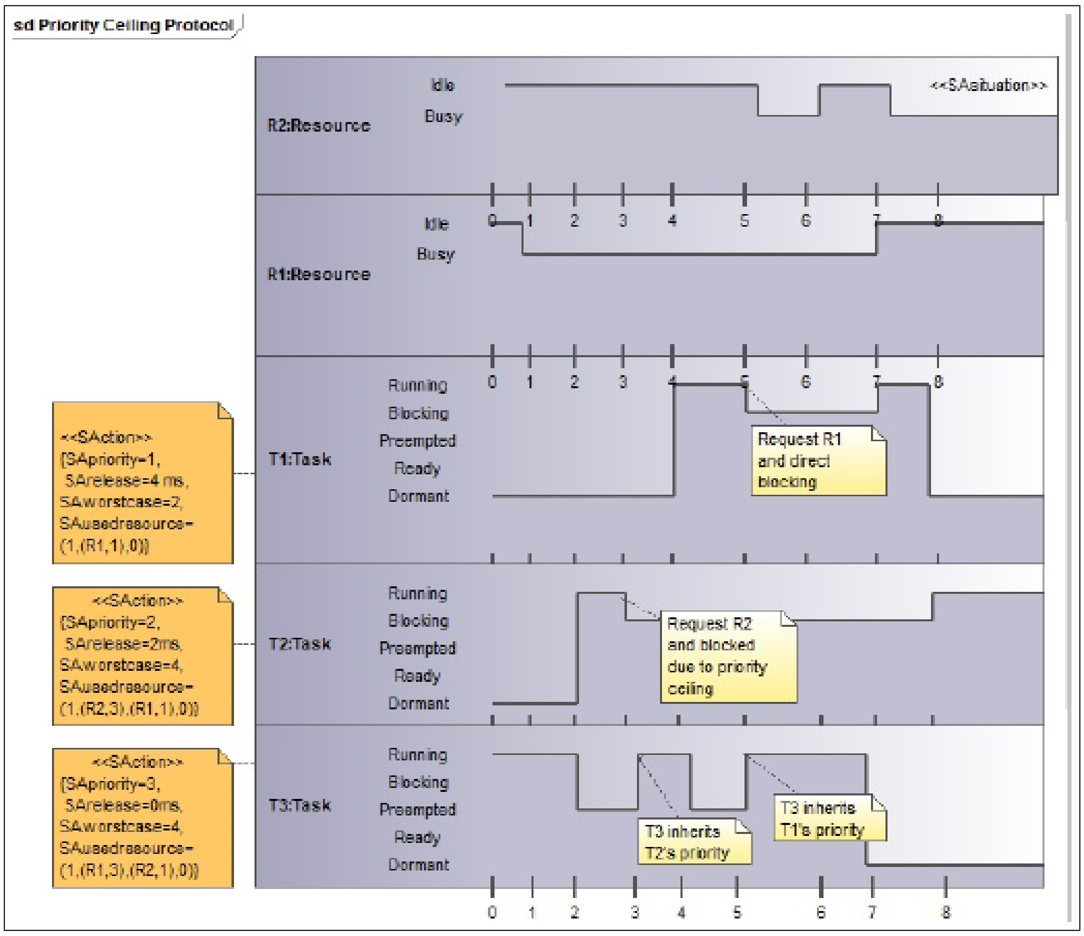

sd Priority Ceiling Protocol
R2:Resource
Idle
Busy
<<SAsituation>>
R1:Resource
Idle
Busy
T1:Task
T2:Task
T3:Task
Running
Blocking
Preempted
Ready
Dormant
0 1 2 3 4 5 6 7 8
<<SAction>> {SApriority=1, SArelease=4 ms, SAworstcase=2, SAusedresource=(1,(R1,1),0)}
<<SAction>> {SApriority=2, SArelease=2ms, SAworstcase=4, SAusedresource=(1,(R2,3),(R1,1),0)}
<<SAction>> {SApriority=3, SArelease=0ms, SAworstcase=4, SAusedresource=(1,(R1,3),(R2,1),0)}
Request R1 and direct blocking
Request R2 and blocked due to priority ceiling
T3 inherits T2's priority
T3 inherits T1's priority

FIGURE 7.5: Timing diagram showing deadlock prevention using PCP

At time 6, T3 releases R2 and at time 7, T3 releases R1. So T3 executes at its inherited priority π(t) (=1) until the time when it releases every resource whose priority ceiling is equal to higher than π(t) (i.e. its inherited priority). T3 completes its execution at time 7.

At that time 7, T1 and T2 are ready. But T1 has higher priority (i.e.,1) it resumes. At time 8, T1 completes its execution and T2 resumes.

7.6.2 SPCP

Further, we have considered Stack Based Priority Ceiling Protocol instead of the Basic Priority Ceiling Protocol for the following reasons-

- Stack Based Priority Ceiling Protocol is simple; the complex priority inheritance rule is not required here.
- No task is ever blocked in Stack Based Priority Ceiling Protocol once its execution begins, so it has a lower context switching overhead.

7.6.2.1 Observation

The sequence diagram (Figure 7.6) shows that deadlock does not occur for this protocol. All the three tasks complete their executions successfully.

The timing diagram (Figure 7.7) shows how deadlock is prevented.

7.6.2.2 Result and discussion

T3 is released at time 0. The ceiling of the system at time 1 is Ω (a non-existing priority level). When T3 requests R1 which is allocated to T3 according to rule 2 (discussed in section 7.2.4). After the allocation of R1, the ceiling of the system is raised to 1, the priority ceiling of R1.

At time 2, T2 is released and it is blocked from starting because the ceiling π'(3)(=1) of the system is higher than the priority of T2. T2's request is denied according to rule 1. This allows T3 to continue execution.

At time 3 T3 requests R2 which is allocated to T3 according to rule 2. When T1 is released at time 4, it cannot start execution according to rule 1.

At time 5, T3 completes execution and releases resources (R1 and R2) and the ceiling of the system is Ω. Consequently, T1 starts execution since it has the highest priority among all the tasks ready at the time. When T1 requests R1, it is allocated to T1.

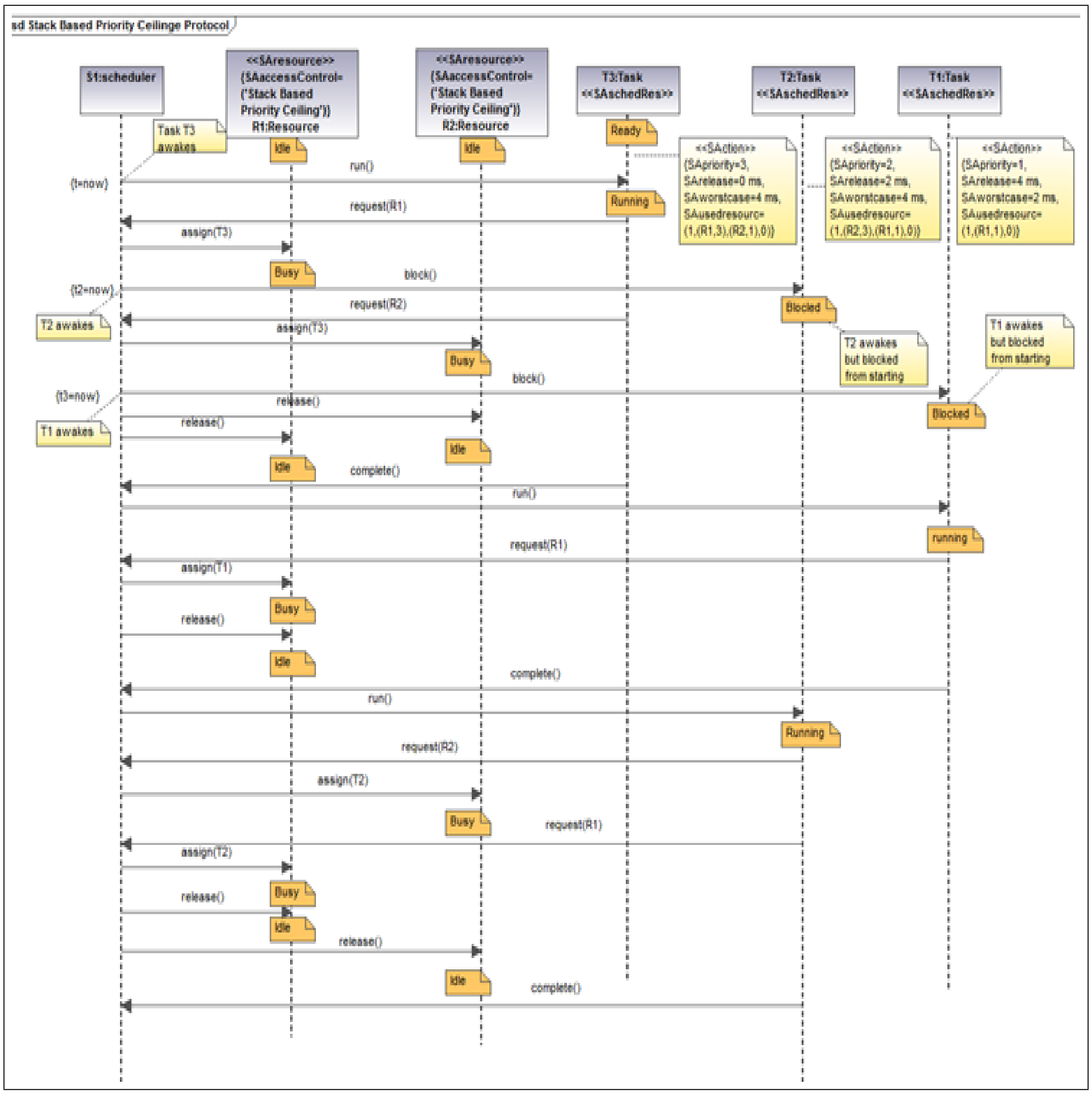

sd Stack Based Priority Ceilinge Protocol
S1:scheduler
<<SAresource>> {SAaccessControl=('Stack Based Priority Ceiling')} R1:Resource
<<SAresource>> {SAaccessControl=('Stack Based Priority Ceiling')} R2:Resource
T3:Task <<SAschedRes>>
T2:Task <<SAschedRes>>
T1:Task <<SAschedRes>>
Task T3 awakes
<<SAction>> {SApriority=3, SArelease=0 ms, SAworstcase=4 ms, SAusedresourc=(1,(R1,3),(R2,1),0)}
<<SAction>> {SApriority=2, SArelease=2 ms, SAworstcase=4 ms, SAusedresourc=(1,(R2,3),(R1,1),0)}
<<SAction>> {SApriority=1, SArelease=4 ms, SAworstcase=2 ms, SAusedresourc=(1,(R1,1),0)}
{t=now}
{t2=now}
{t3=now}
T2 awakes
T1 awakes
T2 awakes but blocked from starting
T1 awakes but blocked from starting
Ready
Running
Blocked
running
Idle
Busy
run()
request(R1)
assign(T3)
block()
request(R2)
release()
complete()
assign(T1)
assign(T2)

Figure 7.6: Sequence diagram showing deadlock prevention using SPCP

T1 completes at time 7 and T2 starts its execution and acquires all the required resources. T2 completes at time 12.

7.6.3 SBPCP

Further, we have considered SBPCP for the following reasons:

- SBPCP avoids paying time or storage overhead.

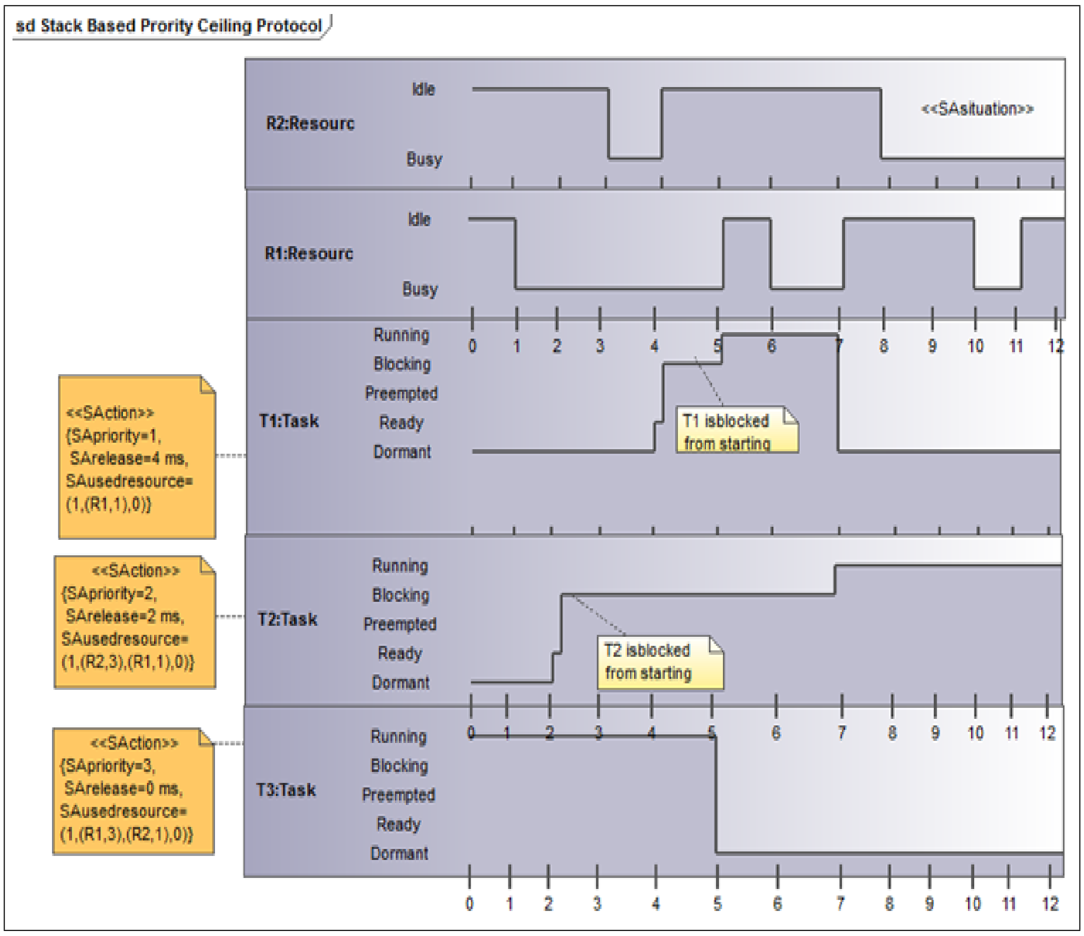

sd Stack Based Prority Ceiling Protocol
R2:Resourc
Idle
Busy
<<SAsituation>>
R1:Resourc
Idle
Busy
T1:Task
Running
Blocking
Preempted
Ready
Dormant
T1 isblocked from starting
<<SAction>> {SApriority=1, SArelease=4 ms, SAusedresource=(1,(R1,1),0)}
T2:Task
T2 isblocked from starting
<<SAction>> {SApriority=2, SArelease=2 ms, SAusedresource=(1,(R2,3),(R1,1),0)}
T3:Task
<<SAction>> {SApriority=3, SArelease=0 ms, SAusedresource=(1,(R1,3),(R2,1),0)}
0 1 2 3 4 5 6 7 8 9 10 11 12

FIGURE 7.7: Timing diagram showing deadlock prevention using SPCP

- Ceilings are defined in terms of preemption levels, instead of priorities, so that the SBPCP applies directly to Earliest Deadline First (EDF) scheduling (without dynamic recomputation of ceilings).
- Potential of resource contentions in dynamic priority systems does not change with time.
- Because there is no blocking after a job starts executing, a stronger EDF schedulability result can be obtained than with dynamic priority ceilings.

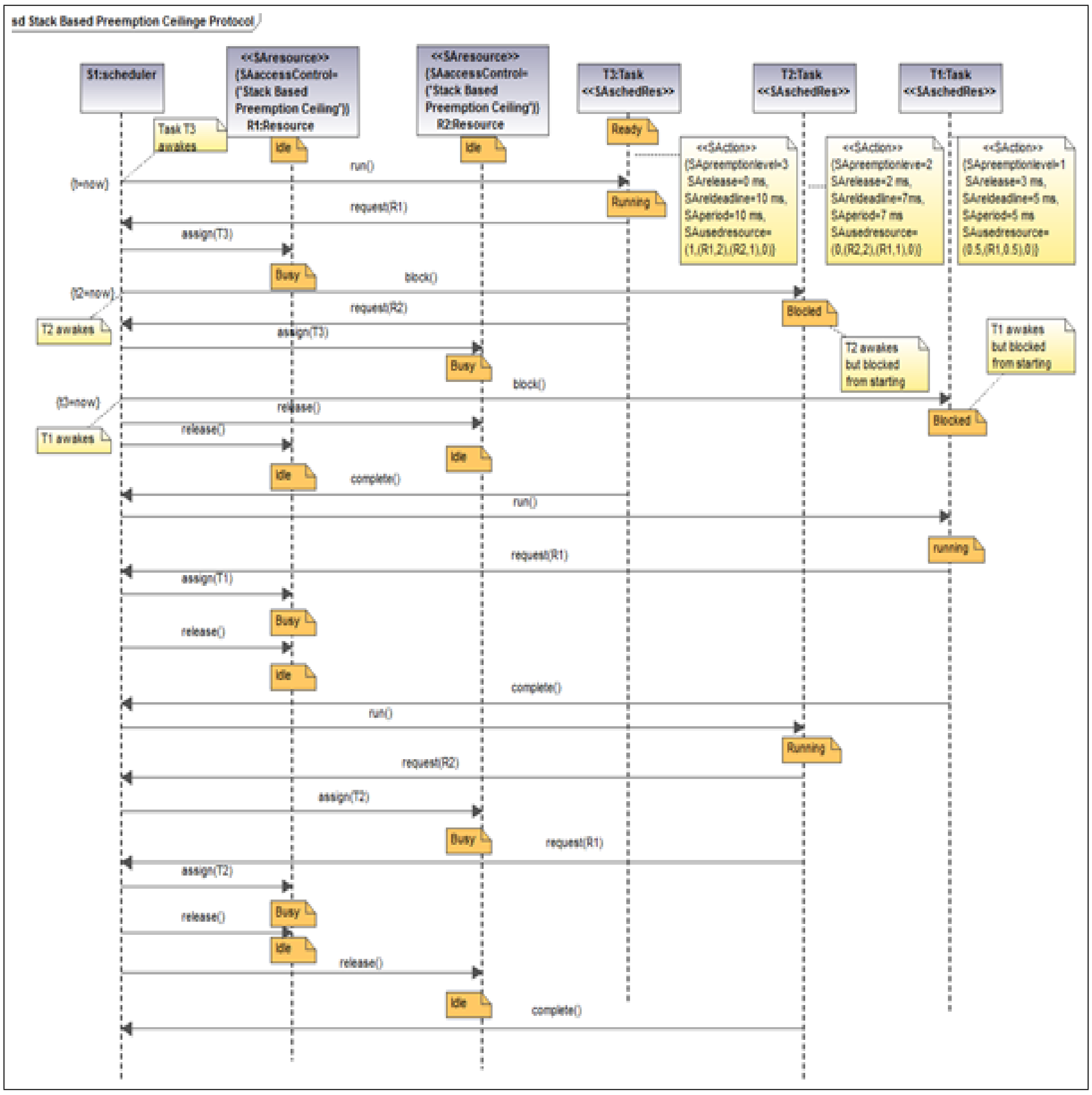

sd Stack Based Preemption Ceilinge Protocol
S1:scheduler
<<SAresource>> {SAaccessControl= ('Stack Based Preemption Ceiling')} R1:Resource
<<SAresource>> {SAaccessControl= ('Stack Based Preemption Ceiling')} R2:Resource
T3:Task <<SAschedRes>>
T2:Task <<SAschedRes>>
T1:Task <<SAschedRes>>
Task T3 awakes
<<SAction>> {SApreemptionlevel=3 SArelease=0 ms, SAreldeadline=10 ms, SAperiod=10 ms, SAusedresource= (1,(R1,2),(R2,1),0)}
<<SAction>> {SApreemptionleve=2 SArelease=2 ms, SAreldeadline=7ms, SAperiod=7 ms SAusedresource= (0,(R2,2),(R1,1),0)}
<<SAction>> {SApreemptionlevel=1 SArelease=3 ms, SAreldeadline=5 ms, SAperiod=5 ms SAusedresource= (0.5,(R1,0.5),0)}
{t=now}
{t2=now}
{t3=now}
T2 awakes
T1 awakes
T2 awakes but blocked from starting
T1 awakes but blocked from starting
Ready
Running
Idle
Busy
Blocked
running
run()
request(R1)
assign(T3)
block()
request(R2)
release()
complete()
assign(T1)
assign(T2)

FIGURE 7.8: Sequence diagram showing deadlock prevention using SBPCP

7.6.3.1 Observation

The sequence diagram (Figure 7.8) shows that deadlock does not occur for this protocol. All the three tasks complete their executions successfully.

The Timing diagram (Figure 7.9) shows how deadlock is prevented.

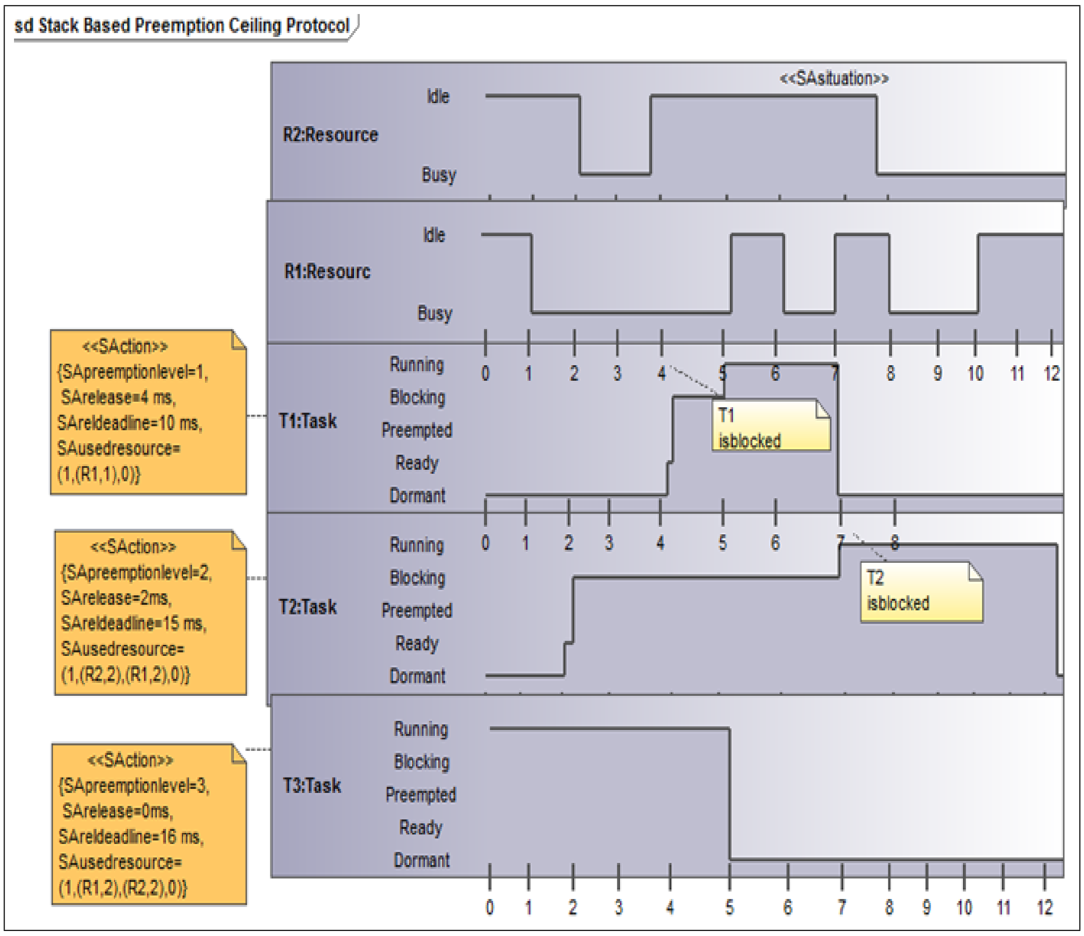

sd Stack Based Preemption Ceiling Protocol
<<SAsituation>>
R2:Resource
Idle
Busy
R1:Resourc
Idle
Busy
T1:Task
Running
Blocking
Preempted
Ready
Dormant
T1 isblocked
T2:Task
T2 isblocked
T3:Task
0 1 2 3 4 5 6 7 8 9 10 11 12
<<SAction>> {SApreemptionlevel=1, SArelease=4 ms, SAreldeadline=10 ms, SAusedresource= (1,(R1,1),0)}
<<SAction>> {SApreemptionlevel=2, SArelease=2ms, SAreldeadline=15 ms, SAusedresource= (1,(R2,2),(R1,2),0)}
<<SAction>> {SApreemptionlevel=3, SArelease=0ms, SAreldeadline=16 ms, SAusedresource= (1,(R1,2),(R2,2),0)}

FIGURE 7.9: Timing diagram showing deadlock prevention using SBPCP

7.6.3.2 Result and discussion

T3 is released and starts execution at time 0. At time 1, T3 requests R1. R1 is allocated to T3 according to Rule 2 in section 7.2.5.

At time 2, T2 is released and it is blocked from starting according to Rule 1. T3 inherits T2's priority and requests R2 which is allocated to T3 at the same time instance according to Rule 2. When T1 is released at time 4, it cannot start execution according to Rule 1. T3 inherits T1's priority and continue execution.

At time 5, T3 completes execution and releases resources (R1 and R2) and the ceiling of the system is Ω. Consequently, T1 starts execution since it has the highest priority among all the tasks ready at that time. When T1 requests for R1, it is allocated to T1.

T1 completes at time 7 and T2 starts its execution and acquires all required resources. T2 completes at time 12.

7.7 Verification of resource access control protocols

In this section, we are considering the formal representation of resource access control protocols utilizing CFG for the automated verification of deadlock occurrence and its prevention. Further, the demonstrated formal portrayals have been confirmed for the rightness of the outline utilizing the ANTLR tool. Next, with the reference to this work, these protocols have been investigated and implemented using JAVA and Python to demonstrate the occurrence of deadlock for PIP and how this can be avoided by using PCP, SPCP and SBPCP.

7.7.1 Formal specification

Table 7.4 shows the formal representation of the PIP, PCP, SPCP and SBPCP.

7.7.1.1 Illustration of the grammar

For the grammar, test_PIP_ps has been considered as the start symbol.

test_PIP_ps : pip;

The formal specification of the four protocols of interest considers the set of active objects of type task, resource and the scheduler and also considers events that are responsible for the interactions among the objects. Events may occur based on some

Table 7.4: Formal specification of resource access control protocols

```
grammar abc.g;
//Parser
options output=AST; backtrack=true;
test_PIP_ps : pip ;
pip : act_obj_type event;
event : 'pipAlgo'|'pcpAlgo'| 'stackPcpAlgo'|'stackPreAlgo';
act_obj_type : task|resource|scheduler;
scheduler : schd_id;
schd_id : SID;
resourceWithstate : resource state ;
state : 'Busy' | 'Idle';
task : t1 t2 t3 ;
t1 : '{'sapriority sarelease saworstcase sausedresource'}';
t2 : '{'sapriority sarelease saworstcase sausedresource'}' ;
t3 : '{'sapriority sarelease saworstcase sausedresource'}' ;
sapriority : INT ;
sarelease : INT ;
saworstcase : INT ;
sausedresource : '(1,' ('('resource',' time'),')+ '0)';
resource : 'R1'|'R2';
time : INT;
//Lexer
INT : ('0'..'9')+;
CHAR : ('A'..'Z'|'a'..'z')+;
SID : ('s'|'S')('0'..'9')+;
RID : ('r'|'R')('0'..'9')+;
TID : ('t'|'T')('0'..'9')+;
ID : ('a'..'z'|'A'..'Z')+('0'..'9')+;
REAL : ('0'..'9')+('.')*('0'..'9')*;
```

specific conditions. Since this research work aims at formally representing the above mentioned four protocols thus only the conditions specified for those four protocols have been considered here.

pip : act_obj_type event;

event : 'pipAlgo'|'pcpAlgo'| 'stackPcpAlgo'|'stackPreAlgo';

act_obj_type : task|resource|scheduler;

The scheduler has an identification number, denoted by, schd_id which is of type SID, defined in the lexer specification section.

scheduler : schd_id;

schd_id : SID;

Each resource has a name denoted by resource and its associated state is denoted by state. For a resource type, the state can be either Busy or Idle.

resourceWithstate : resource state ;

state : 'Busy' | 'Idle';

Here a set of three tasks t1, t2 and t3 have been considered for simplicity. Each task has its associated priority, release time, computation time denoted by sapriority, sarelease, saworstcase respectively. Each of these is of type INT, defined in the lexer specification part. The tasks may also have a set of used resources for execution which is denoted by sausedresource. Here, two resources named R1 and R2 have been considered given by the production,

resource : 'R1'|'R2';

While specifying used resources, the duration of the time for which these resources have been used must be specified and is denoted by "time" which is of type INT.

task : t1 t2 t3 ;

t1 : '{'sapriority sarelease saworstcase sausedresource'}';

t2 : '{'sapriority sarelease saworstcase sausedresource'}';

t3 : '{'sapriority sarelease saworstcase sausedresource'}';

sapriority : INT ;
sarelease : INT ;
saworstcase : INT ;

Table 7.5: Production rules with associated action statements

```
pip returns[String param, String algo]:act_obj_type{$param= $act_obj_type.finalString;}
event{$algo = $event.text;};
event : ’pipAlgo’|’pcpAlgo’| ’stackPcpAlgo’|’stackPreAlgo’;
act_obj_type returns[String finalString]task{String str1=”name:T3@release:” +
Integer.toString($task.t3releasetime) + ”@currentPriority:” +
Integer.toString($task.t3priority) + ”@state:READY@resource:” +$task.t3usedresouce;
......
$ finalString = str1+’;’+str2+’;’+str3;}|resource|scheduler;
```

sausedresource : ’(1,’ (’(’resource’,’ time’),’)+ ’0)’;

resource : ’R1’|’R2’;

time : INT;

For the sake of simplicity in the implementation, this grammar considers only three tasks. It can be further extended to include a set of tasks by slightly modifying the grammar.

7.7.2 Implementation

Those four resource access control protocols have been implemented and linked to the developed grammar in a form accepted by ANTLR tool to show the schedulability of these two protocols. To encode the given algorithms Python language has been used for ease of implementation. Each production rule has been associated with some action statements that help to return the intermediate results at each step. Some portions of the grammar production rules with associated action statements are shown in Table 7.5.

7.7.3 Result analysis

Some portions of the output of the developed python programs for those protocols are given in Table 7.6 and Table 7.7. Hypothetically, it is known that Priority

Table 7.6: Output of the Priority Inheritance Protocol

```
T3 READY
Schedular—-run()————→ T3
T3 RUNNING
Schedular←——request(R1)——T3
Schedular——assign(R1)————→T3
R1———-→ BUSY
==================================
Schedular———-preempt()——T3
Schedular ——Run() ——T2
T2 RUNNING
Schedular←——Request(R2)——T2
Schedular——assign(R2)——→T2
R2—-→ BUSY
==================================
Schedular——preempt()——T2
Schedular——Run()——T1
T1 RUNNING
Schedular ←——Request(R1)——T1
Schedular——block()———→ T1
T3——inherits——T1′s priority
Schedular——run()——→T3
T3 RUNNING
==================================
Schedular←——Request(R1)——T3
Schedular——assign(R1)——→T3
R1—-→BUSY
==================================
Schedular ←——Request(R2)——T3
Schedular——block()———-→T3
T2——inherits——T3′s priority
Schedular——run()———→T2
T2 RUNNING
==================================
Schedular←——Request(R2)——T2
Schedular——assign(R2)———-→T2
R2———-→ BUSY
==================================
Schedular ←———-Request(R1)——T2
Schedular——block()———-→T2
T3——inherits——T2′s priority
Schedular——run()———- →T3
T3 RUNNING
==================================
————DEADLOCK—————
```

Table 7.7: Output of the Stack Based Priority Ceiling Protocol

```
T3 READY
Schedular—-run()——→T3
T3 RUNNING
Schedular←——request(R1)——T3
Schedular——assign(R1)——→T3
R1—-→BUSY
=====================================
Schedular—-block()——→T2
=====================================
Schedular—-block()——→T1
=====================================
Schedular——release()——→R1
Schedular←——complete()——T3
=====================================
Schedular—-run()——→T1
T1 RUNNING
Schedular←——request(R1)——T1
Schedular——assign(R1)——→T1
R1—-BUSY
Schedular——release()——→R1
Schedular←——complete()——T1
=====================================
Schedular—-run()——→T2
T2 RUNNING
Schedular←——request(R2)——T2
Schedular——assign(R2)——→T2
R2—-→BUSY
=====================================
Schedular——release()——→R2
Schedular←——complete()——T2
=====================================
```

Inheritance Protocol may hinder the execution of the tasks and results in a deadlock whereas, in the case of PCP, SPCP and SBPCP, all tasks will be executed successfully to completion without causing a deadlock situation.

7.8 Comparison with related work

Maria et al. discussed the task scheduling capabilities of UML and its profiles in RTS [131]. Their work is focused primarily on the priority inheritance protocol but analysis of the protocol regarding resource contention of deadlock prevention is outside the scope. Authors in [136] presents a formalized and mechanically checked verification for the rightness of only priority inheritance protocol. In comparison to those research works, this Chapter formally analyses the existing resource access control protocols (Priority Inheritance, Priority Ceiling, Stack Based Priority Ceiling and Stack Based Preemption Ceiling Protocols) using sequence and timing diagrams to study deadlock in real-time systems.

7.9 Threats to validity of proposed approach

This research work emphasizes on the issues related to deadlock and deadline of resource access control protocols. The Priority Inheritance Protocol is used for sharing critical resources but it does not prevent deadlock if nested critical sections are present. This shortcoming is represented using one UML model. Further, the Priority Ceiling, Stack Based Priority Ceiling and Stack Based Preemption Ceiling Protocols are used to overcome the difficulty using other improved models. We have analyzed those protocols using a limited number of data set. However, model behavior may change with increasing data set values.

7.10 Conclusion

In the last few years, real-time processing seems to be the essential part of an operating system and the scheduling of RTS is an important area of research in today's life.

This dissertation explores the comparison among the various resource access protocols for uniprocessor RTS. This chapter concentrates on the occurrence of deadlock in the Priority Inheritance Protocol and the prevention of such using Priority Ceiling Protocol, Stack Based Priority Ceiling Protocol and Stack Based Preemption Ceiling Protocol. Using UML/SPT based Sequence and Timing Diagrams, we model the above mentioned protocols. Further, we analyze these to highlight deadlock occurrence and deadlock avoidance.

Chapter 8

Conclusion

8.1 Conclusion

RTSS are deployed nowadays in many applications, including home appliances, automotive, avionics, military applications etc. These systems should be logically correct and should satisfy a set of timing constraints. In addition, they are reactive and concurrent to handle the different concurrent events that come from the environment in which these systems are deployed. Such characteristics make the design of RTSS complex and challenging.

In this dissertation, we have presented some new ideas and developed some novel approaches for modeling, analysis and verification of RTSS through the different phases of development. We have considered some of the commonly used UML diagrams to develop our methodology.

Modeling is an important engineering activity, which relies on using models to raise the abstraction level. This widens the engineers' visibility and increases their control over the complexity of the systems they are building or managing. The model-driven approach is therefore adequate to address the complex issues of RTSS. It is very advantageous to use models with rigorous semantics. This enables the verification of design and potentially the automatic synthesis of implementations.

The design specification for RTSS modeled using UML diagrams presents different aspects of the system, some of which represent overlapping characteristics. Hence, risks of inconsistencies between such related design models are inevitable. We have defined a set of conditions for ensuring consistency between the UML models. A formal representation with semantic actions has been defined for representing four UML diagrams, which are mostly used in modeling. The consistency rules have been verified using the formal representation of UML diagrams depicting design.

It is seen that most of the errors detected at the later stage of any software development are caused due to inconsistency between requirements and design. This leads to huge project cost for correction and sometimes schedule slippages. Large-scale software developments often fail as designers are not able to produce complete, understandable, unambiguous and traceable design documents from the functional requirements phase. A prime concern of RTSS is that all the requirements related to time must be traced in all the phases of the software development life cycle consistently. A comprehensive framework is developed that helps in the verification of requirements in design. This framework is based on the formal design with semantic constructs that verifies the timing constraints through metrics-based approach.

Due to the concurrent nature of RTSS, there may be contention for resources that requires scheduling (i.e. how tasks are granted access to a given resource). An important problem that arises in the context of such RTS is the effect of blocking which occurs due to the need for synchronization of tasks that share common logical or physical resources. We have compared various resource access control protocols using model-based approach.

Modeling, analysis and verification are significant activities in the development cycle of any RTSS, which consumes a considerable amount of overall effort. The methods and frameworks presented in this dissertation provide a firm foundation towards a significant reduction of designing effort and cost for the software industry. Our derived framework guides practitioners in their application of UML/MARTE in industrial contexts. This will help practitioners bridge the gap between modeling standards and the modeling needs of industrial RTSS.

8.2 Future work

- The metamodeling approach used for the definition of UML profile in general and which we used for the extension of UML/MARTE presents an interesting issue. Indeed, the concepts required for a certain domain could be modeled in a variety of manners, which leads to different metamodels. Therefore, it is necessary to assess the consistency between the different profiles and extensions.
- Our approach on using schedulability analysis to model, analyze and verify different resource access control protocols provides limited feedback to the designer. Other improtant questions need to be addressed such as when the analysis shows that various requirements are not consistent, what can be done to fix the inconsistency? Is it possible to provide more fine-grained feedback in pointing out the origin of the inconsistency? What changes can be operated on the design model that might fix the problem?
- Our framework concentrates on the requirements analysis and design phases of a software development life cycle. In future, we plan to consider other stages of development life cycle like tracing requirements to implementation and develop testing techniques for RTSS.

Appendix A

Table A.1: Formal specification for collecting information from Use case diagram

```
//Parser
grammar usecase;
options {language = Java; output = AST; backtrack=true;}
scope global{
   static String str;   //storing use case ids
   static ArrayList usecase;   //List for storing use cases
   static ArrayList uactor;   //List for storing actors of each use case
   static ArrayList ustereo;   //List for storing stereotypes of each use case
   static ArrayList timing_constraint;   //List for storing timing constraints of each use case
   static HashMap hm;   //Storing all the actors, stereotypes and timing constraints for all use cases
}
usecase_diagram:(use_case_dia_id {      //Unique id for each Use case diagram
   global_scope.usecase=new ArrayList();
}) use_case+ actor* uc_relation* actor_relation*;
use_case: (UC_id {
   global_scope.str=$UC_id.text;
   global_scope.uactor=new ArrayList();
   global_scope.ustereo=new ArrayList();
   global_scope.timing_constraint=new ArrayList();
   global_scope.hm = new HashMap();
   if(!global_scope.usecase.contains($UC_id.text))   //Checking uniqueness of use cases
      global_scope.usecase.add($UC_id.text);   //Adding use cases in the list
}) uc_name uannotation;
uc_name : CHAR;
uannotation: ustereotype* utiming_cons*;
utiming_cons: (ucons_id {
   //Semantic actions
if(!global_scope.timing_constraint.contains($ucons_id.text)) //Checking uniqueness of timing constraints
      global_scope.timing_constraint.add($ucons_id.text);      //Adding timing constraints in the list
         global_scope.hm.put(global_scope.str+”_TC”, global_scope.timing_constraint);   //Associates a use case id with its timing constraints
})ucon_type ucons_desc;
ucons_type : 'delay'|'duration'|'deadline';
ustereotype : (ustreo_id {
   //Semantic actions
if(!global_scope.ustereo.contains($ustereo_id.text))      //Checking uniqueness of stereotypes
global_scope.ustereo.add($ustereo_id.text);   //Adding stereotypes in list
global_scope.hm.put(global_scope.str+”_ST”, global_scope.ustereo);      //Associates a use case id with its stereotypes
}) stereotype_name {utag+};
utag:utag_name utag_value;
stereotype_name :CHAR;
stereotype_name:'<<'CHAR'>>';
utag_name:CHAR;
utag_value:CHAR;
ucons_desc: CHAR;
uc_relation: UC_id UC_reltype UC_id;
actor_relation: actor_id Actor_reltype actor_id;
actor : (actor_id {
   //Semantic actions
if(!global_scope.uactor.contains($actor_id.text))      //Checking uniqueness of actors
global_scope.uactor.add($actor_id.text);      //Adding actors in list
global_scope.hm.put(global_scope.str+”_AT”, global_scope.uactor);
         //Associates a use case id with its actors
}) actor_name;
actor_id : AID;
actor_name : CHAR;
ucons_id: CID;
ustreo_id:STID;
//Lexer
use_case_dia_id:CHAR;
UC_reltype: 'association'|'<<include>>'|'<<extend>>'|'generalization';
Actor_reltype:'generalization';
CID : ('C'|'c')('0'..'9')+ ;
AID : ('A'|'a')('0'..'9')+ ;
UC_id : ('U'|'u')('0'..'9')+ ;
STID : ('ST'|'st')('0'..'9')+ ;
CHAR : ('0'..'9'|'A'..'Z'|'a'..'z'|'≤'|'_')+ ;
WS : ( ' ' | '\t' | '\r' | '\n' ){$channel=HIDDEN;};
```

Table A.2: Formal specification for consistency checking from Sequence diagram

```
//Parser
grammar seq_dia;
options {language = Java; output = AST; backtrack=true;}
scope global{
  static String seq_diagram; //Storing sequence diagram id
  static ArrayList seq_actor;
  static ArrayList seq_stereo;
  static ArrayList seq_usecase;
  static ArrayList seq_timing_constraint;
  static ArrayList seq_lifeline;
  static ArrayList seq_par;
  static ArrayList seq_method;
  static ArrayList seq_message;
}
sequence_diagram: (seq_id {
global_scope.seq_diagram=$seq_id.text;
global_scope.seq_usecase=new ArrayList();     //Lists for storing usecase id
global_scope.seq_actor=new ArrayList();     //Lists for storing actors
global_scope.seq_stereo=new ArrayList();   //Lists for storing stereotypes
global_scope.seq_timing_constraint=new ArrayList();   //Lists for storing timing constraints
global_scope.seq_lifeline= new ArrayList();   //Lists for storing lifelines
global_scope.seq_par= new ArrayList();   //Lists for storing participants
global_scope.seq_method= new ArrayList();   //Lists for storing methods
global_scope.seq_message= new ArrayList();   //Lists for storing messages
}) (uc_id {
  if(global_scope.usecase.contains($uc_id.text)) //Try to find out if the current token exists in use case
    global_scope.seq_usecase.add($uc_id.text);   //Adding use case id in list
} )life_line+ seq_annotation;
uc_id : UID;
seq_id : SID;
life_line:participant | actor;
participant : (par_id {
if(!global_scope.seq_par.contains($par_id.text))
global_scope.seq_par.add($par_id.text);}) par_name edge+;
actor: (actor_id {
    //Semantic actions
  if(global_scope.uactor.contains($actor_id.text))     //Try to find out if the current token exists in use case
if(!global_scope.seq_actor.contains($actor_id.text))   //Checking uniqueness of actors
global_scope.seq_actor.add($actor_id.text);   //Adding actors in list
} ) actor_name edge+;
seq_lifeline.addAll(seq_actor);
seq_lifeline.addAll(seq_par);
seq_annotation: seq_stereotype* seq_timing_cons*;
seq_stereotype : (seq_streo_id {
  //Semantic actions
if(global_scope.ustereo.contains($seq_stereo_id.text))
//Try to find out if the current token exists in use case
    if(!global_scope.seq_stereo.contains($seq_stereo_id.text))   //Checking uniqueness of stereotypes
global_scope.seq_stereo.add($seq_stereo_id.text);   //Adding stereotypes in list
}) stereotype_name {seq_tag+};
seq_tag: seq_tag_name seq_tag_value;
seq_tag_name: CHAR;
seq_tag_value:CHAR;
seq_timing_cons: (seq_cons_id {     //Semantic actions
if(global_scope.timing_constraint.contains($seq_cons_id.text))
  //Try to find out if the current token exists in use case
    if(!global_scope.seq_timing_constraint.contains($seq_cons_id.text))
  //Checking uniqueness of timing constraints
global_scope.seq_timing_constraint.add($seq_cons_id.text);   //Adding timing constraints in list
}) seq_type seq_desc;
seq_type : 'delay'|'duration'|'deadline';
par_id : PID ;
par_name : CHAR ;
seq_event : event_id event_name Event_type event_time ;
```

```
event_id: EID;
event_name: CHAR;
val :INT|REAL ;
stmt: edge|CHAR;
edge: event_src map_func event_dest;
event_src : seq_event;
event_dest : seq_event;
map_func : msg_func|method_func;
msg_func : (msg_ID {
if (!global_scope.seq_message.contains($msg_id.text))    //Checking uniqueness of messages
global_scope.seq_message.add($message_id.text);    //Adding messages in list
}) msg_name;
msg_ID : MSID;
msg_name : CHAR;
actor_name: CHAR;
method_func : (method_ID {
if (!global_scope.seq_method.contains($method_id.text))    //Checking uniqueness of methods
global_scope.seq_method.add($method_id.text);    //Adding methods in list
}) method_name;
method_ID : MTID;
method_name : CHAR;
actor_id:AID;
seq_cons_id:CID;
seq_streo_id:STID;
stereotype_name :'<<'CHAR'>>';
seq_tag_value:CHAR;
cons_desc : '(' time_diff_event relation dur ')';
time_diff_event: start_event_time alg_op end_event_time;
start_event_time : event_time;
end_even_time : event_time;
alg_op : '+' |'-' |'*' |'/' ;
relation : '<' |'>' | '=' |'≥' |'≤';
dur: INT ;
event_time : INT ;
Event_type : 'send'| 'recv';
//Lexer
AID : ('A'|'a')('0'..'9')+ ;
UID : ('U'|'u')('0'..'9')+ ;
PID : ('P'|'p')('0'..'9')+ ;
EID : ('E'|'e')('0'..'9')+ ;
SID : ('S'|'s')('0'..'9')+ ;
SCID : ('C'|'c')('0'..'9')+ ;
STID : ('ST'|'st')('0'..'9')+ ;
MSID : ('MS'|'ms')('0'..'9')+ ;
MTID : ('MT'|'mt')('0'..'9')+ ;
CHAR : ('A'..'Z'|'a'..'z'|'≤'|'_')+ ;
WS : ( ' ' | '\t' | '\r' | '\n' ){$channel=HIDDEN;};
INT : ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')+ ;
```

TABLE A.3: Formal specification for consistency checking from Timing diagram

```
//Parser
grammar timing;
options output = AST; backtrack=true;
scope global{
   static String timing_diagram;    //Storing timing diagram id
   static String t_seq_dia;    //storing sequence diagram id
   static ArrayList par;    //List for storing participants
   static ArrayList par_method;    //List for storing methods for each partcipants
   static ArrayList par_message;    //List for storing messages for each partcipants
   static ArrayList stereo;    //List for storing stereotypes
   static ArrayList timing_constraint;    //List for storing timing constraints
}
timing_diagram: (time_id {
   //Semantic actions
   global_scope.timing_diagram=$time_id.text;
   global_scope.par=new ArrayList();
   global_scope.par_method=new ArrayList();
   global_scope.par_message=new ArrayList();
   global_scope.stereo=new ArrayList();
   global_scope.timing_constraint=new ArrayList();})
   (seq_id {
   if(global_scope.seq_diagram.equals($seq_id.text))
//Checking if the current token exists in sequence diagram
   global_scope.t_seq_dia=$seq_id.text;}) //Adding sequence diagram id in the list
participant+ time annotation;
participant: (par_id {
   //Semantic actions
   if(!global_scope.par.contains($par_id.text))    //Checking uniqueness of participants
      global_scope.par.add($par_id.text);    //Adding participants in list
      System.out.println("Array List Participant Values"+ $par_id.text);
//Displaying participant id
}) par_name state+ event+;
time_id: TID;
par_id : PCID;
par_name : CHAR;
time :DIGIT;
state: state_ID state_name;
state_ID : SID;
state_name : CHAR;
event: event_id event_name map_func? state_trans? ;
event_id : EID;
event_name : CHAR;
map_func : msg | mthd annotation;
msg:(msg_id {
if(global_scope.seq_message.contains($msg_id.text))
//Checking if the current token exists in sequence diagram
if(!global_scope.par_message.contains($mthd_id.text)) //Checking uniqueness of messages
global_scope.par_message.add($mthd_id.text);
})msg_name;
mthd : (mthd_id {
if(global_scope.seq_method.contains($mthd_id.text))
//Checking if the current token exists in sequence diagram
   if(!global_scope.par_method.contains($mthd_id.text))    //Checking uniqueness of methods
      global_scope.par_method.add($mthd_id.text);    //Adding methods in list
      System.out.println("Array List Method Values "+ $mthd_id.text); //Displaying method id
}) mthd_name;
```

```
msg_id : MSID;
msg_name : CHAR;
mthd_id : MTID;
mthd_name: CHAR;
annotation: stereotype* timing_cons*;
stereotype : (stereo_id {
   //Semantic actions
   if(!global_scope.stereo.contains($stereo_id.text))
   //Checking uniqueness of stereotypes
      global_scope.stereo.add($stereo_id.text);    //Adding stereotypes in list
      System.out.println("Array List stereotype Values"+ $mthd_id.text);
//Displaying stereotype id
}) stereotype_name '{'tag+'}'; tag:tag_name tag_value;
streo_id: STID;
stereotype_name :'<<'CHAR'>>';
tag_name: CHAR;
tag_value : CHAR;
timing_cons: (cons_id {
   //Semantic actions
   if(!global_scope.timing_constraint.contains($cons_id.text))
      //Checking uniqueness of timing constraints
      global_scope.timing_constraint.add($cons_id.text);
//Adding timing constraints in the list
      System.out.println("Array List timing_constraint Values "+ $cons_id.text);
//Displaying timing_constraint id
}) cons_type cons_desc ;
cons_id: TCID;
cons_type : 'delay'|'duration'|'deadline' ;
cons_desc: CHAR ;
state_trans: par_id from_state to_state;
from_state: state;
to_state: state;
//lexer
TID : ('T'|'t')('0'..'9')+ ;
PCID : ('PC'|'pc')('0'..'9')+ ;
SID : ('S'|'s')('0'..'9')+ ;
EID : ('E'|'e')('0'..'9')+ ;
MSID : ('MS'|'ms')('0'..'9')+ ;
MTID : ('MT'|'mt')('0'..'9')+ ;
TCID : ('TC'|'tc')('0'..'9')+ ;
STID : ('ST'|'st')('0'..'9')+ ;
TTID : ('TT'|'tt')('0'..'9')+ ;
DIGIT : ('0'..'9')+ ;
CHAR : ('A'..'Z'|'a'..'z'|'≤'|'_')+ ;
WS : ( ' ' | '\t' | '\r' | '\n' ){$channel=HIDDEN;};
```

Table A.4: Formal specification for consistency checking from Class diagram

```
//Parser
grammar class_dia;
options language = Java; output = AST; backtrack=true;
scope global{
static String c_timing_diagram;    //Storing Timing diagram id
static ArrayList class;    //List for storing classes
static ArrayList c_method;    //List for storing methods
static ArrayList c_stereo;    //List for storing stereotypes
static ArrayList c_timing_constraint;    //List for storing timing constraints
}
class_diagram:(class_dia_id {    //Unique id for Class diagram
  global_scope.class=new ArrayList();
  global_scope.c_method=new ArrayList();
  global_scope.c_stereo=new ArrayList();
  global_scope.c_tim_con=new ArrayList();
}) (td_id {
  if(global_scope.timing_diagram.equals($td_id.text))
//Try to find out if the current token exists in Timing diagram
    global_scope.c_timing_diagram=$td_id.text;    //Adding Timing diagram id
}) classes+ relation*;
td_id: TID;
classes: (class_id {
  //Semantic actions
  if(global_scope.par.contains($class_id.text))
//Try to find out if the current token exists in Timing diagram
    if(!global_scope.class.contains($class_id.text))        //Checking uniqueness of classes
    global_scope.class.add($class_id.text);        //Adding classes in list
}) cname method_class* attribute* c_annotation;
cname: CHAR;
class_dia_id:CID; attribute: Access_specifier? Data_type attribute_name;
attribute_name: CHAR;
method_class: (method_ID {
  //Semantic actions
  if(global_scope.par_method.contains($method_id.text))
//Try to find out if the current token exists in Timing diagram
    if(!global_scope.c_method.contains($method_id.text))        //Checking uniqueness of methods
      global_scope.c_method.add($method_id.text);        //Adding methods in list
}) Access_specifier Data_type method_name '('parameter_list')' c_annotation;
c_annotation: c_stereotype* c_timing_cons*;
c_stereotype : (c_streo_id {
  //Semantic actions
if(global_scope.stereo.contains($c_stereo_id.text))
//Try to find out if the current token exists in Timing diagram
if(!global_scope.c_stereo.contains($c_stereo_id.text))
//Checking uniqueness of stereotypes
      global_scope.c_stereo.add($c_stereo_id.text);        //Adding stereotypes in list
}) stereotype_name '{'c_tag+'}';
c_tag: c_tag_name c_tag_value;
c_tag_name: CHAR;
c_tag_value:CHAR;
c_timing_cons: (c_cons_id {
  //Semantic actions
if(global_scope.timing_constraint.contains($c_cons_id.text))
//Try to find out if the current token exists in Timing diagram
if(!global_scope.c_tim_con.contains($c_cons_id.text))
//Checking uniqueness of timing constraints
      global_scope.c_tim_con.add($c_cons_id.text);        //Adding timing constraints in list
}) c_type c_desc ;
```

```
c_streo_id: STID;
stereotype_name :'<<'CHAR'>>';
class_id:PCID;
c_cons_id: TCID ;
c_type : 'delay'|'duration'|'deadline' ;
c_desc: CHAR ;
method_ID : MTID;
method_name:CHAR;
parameter_list: parameter* ;
parameter: Data_type parameter_name ;
parameter_name: CHAR ;
relation: cname multiplicity* cname relationship ;
relationship: class_id description Type | class_id Type;
multiplicity: DIGIT '..' DIGIT ;
Access_specifier: '+' |'-'| '#';
description: CHAR;
Data_type: CHAR;
Type: 'aggregation' |'association' |'generalization' ;
//lexer
TID : ('T'|'t')('0'..'9')+ ;
CID : ('C'|'c')('0'..'9')+ ;
PCID : ('PC'|'pc')('0'..'9')+ ;
MTID : ('MT'|'mt')('0'..'9')+ ;
TCID : ('TC'|'tc')('0'..'9')+ ;
STID : ('ST'|'st')('0'..'9')+ ;
DIGIT : ('0'..'9')+ ;
CHAR : ('A'..'Z'|'a'..'z'|'≤'|'_') ;
WS : ( ' ' | '\t' | '\r' | '\n' ){$channel=HIDDEN;};
```

Bibliography

[1] OCL 1.5 Specification. http://www.omg.org.

[2] UML 2.0 Specification. http://www.omg.org.

[3] Netta Aizenbud-Reshef, Richard F Paige, Julia Rubin, Yael Shaham-Gafni, and Dimitrios S Kolovos. Operational semantics for traceability. In *ECMDA Traceability Workshop (ECMDA-TW)*, pages 7–14, 2005.

[4] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[5] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[6] Ludovic Apvrille, J-P Courtiat, Christophe Lohr, and Pierre de Saqui-Sannes. TURTLE: A real-time UML profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering*, 30(7):473–487, 2004.

[7] Ludovic Apvrille, Pierre de Saqui-Sannes, and Ferhat Khendek. TURTLE-P:A UML profile for the formal validation of critical and distributed systems. *Software and Systems Modeling*, 5(4):449–466, 2006.

[8] Theodore P Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[9] Cesare Bartolini, Antonia Bertolino, Guglielmo De Angelis, and Giuseppe Lipari. A UML Profile and a methodology for real-time systems design. In *Software Engineering and Advanced Applications, 2006. SEAA 06. 32nd EUROMICRO Conference on*, pages 108–117. IEEE, 2006.

[10] M Encarnación Beato, Manuel Barrio-Solórzano, Carlos E Cuesta, and Pablo de la Fuente. UML automatic verification tool with formal methods. *Electronic Notes in Theoretical Computer Science*, 127(4):3–16, 2005.

[11] Simona Bernardi and Dorina C Petriu. Comparing two UML Profiles for non-functional requirement annotations: the SPT and QoS Profiles. *SVERTS, Lisbone, Portugal, October*, 2004.

[12] Lutz Bichler, Ansgar Radermacher, and Andreas Schurr. Evaluating UML extensions for modeling real-time systems. In *Object-Oriented Real-Time Dependable Systems, 2002.(WORDS 2002). Proceedings of the Seventh International Workshop on*, pages 271–278. IEEE, 2002.

[13] Alessandro Biondi, Giorgio C Buttazzo, and Marko Bertogna. Schedulability analysis of hierarchical real-time systems under shared resources. *IEEE Transactions on Computers*, 65(5):1593–1605, 2016.

[14] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984.

[15] Grady Booch. *Object oriented analysis & design with application.* Pearson Education India, 2006.

[16] Alan Burns, Marina Gutierrez, Mario Aldea Rivas, and Michael Gonzalez Harbour. A deadline-floor inheritance protocol for EDF scheduled embedded real-time systems with resource sharing. *IEEE Transactions on Computers*, 64(5):1241–1253, 2015.

[17] Laura Florentina Cacovean and Florin Stoica. CTL model update implementation using ANTLR tools. In *WSEAS International Conference. Proceedings. Recent Advances in Computer Engineering*, number 13. WSEAS, 2009.

[18] Laura Florentina Cacovean, Florin Stoica, and Dana Simian. A new model checking tool. In *Proceedings of the European Computing Conference (ECC '11), Paris, France*, pages 358–364, 2011.

[19] Andreu Carminati, Rômulo Silva de Oliveira, Luís Fernando Friedrich, and Rodrigo Lange. Implementation and Evaluation of the Synchronization Protocol Immediate Priority Ceiling in PREEMPT-RT Linux. *Journal of Software*, 7(3):516–525, 2012.

[20] Jayeeta Chanda, Ananya Kanjilal, Sabnam Sengupta, and Swapan Bhattacharya. Traceability of requirements and consistency verification of UML Use case, Activity and Class diagram: A Formal approach. In *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*, pages 1–4. IEEE, 2009.

[21] Marsha Chechik and John Gannon. Automatic analysis of consistency between requirements and designs. *IEEE Transactions on Software Engineering*, 27(7):651–672, 2001.

[22] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, 1990.

[23] Zhenbang Chen, Zhiming Liu, Anders P Ravn, Volker Stolz, and Naijun Zhan. Refinement and verification in component-based model-driven design. *Science of Computer Programming*, 74(4):168–196, 2009.

[24] Jinho Choi, Eunkyoung Jee, and Doo-Hwan Bae. Timing consistency checking for UML/MARTE behavioral models. *Software Quality Journal*, 24(3):835–876, 2016.

[25] Jane Cleland-Huang, Carl K Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003.

[26] Jane Cleland-Huang, Raffaella Settimi, Chuan Duan, and Xuchang Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 135–144. IEEE, 2005.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2009.

[28] Miguel A de Miguel. General framework for the description of QoS in UML. In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 61–68. IEEE, 2003.

[29] Shouvik Dey and Swapan Bhattacharya. Formal Specification of Structural and Behavioural Aspects of Design Patterns. *Journal of Object Technology*, 9(6):99–126, 2010.

[30] Brian Dobing and Jeffrey Parsons. Dimensions of UML diagram use: a survey of practitioners. *Journal of Database Management*, 19(1):1, 2008.

[31] Bruce P Douglass. Designing Real-Time Systems with UML-Part II. *Embedded Systems Programming*, 11:42–65, 1998.

[32] Bruce Powel Douglass. *Real time UML: advances in the UML for real-time systems.* Addison-Wesley Professional, 2004.

[33] Steve Easterbrook and Marsha Chechik. Automated paraconsistent reasoning via model cheking. In *Proceedings of IJCAI Inconsistency Workshop*, volume 8, Aug 2001.

[34] Alexander Egyed. Scalable consistency checking between diagrams-The VIEWINTEGRA Approach. In *Automated Software Engineering, 2001.(ASE*

2001). Proceedings. 16th Annual International Conference on, pages 387–390. IEEE, 2001.

[35] Alexander Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, 2003.

[36] Alexander Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011.

[37] Alexander Egyed and Paul Grunbacher. Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help. *IEEE Software*, 21(6):50–58, 2004.

[38] Mohammed El Shobaki. *On-chip monitoring for non-intrusive hardware/software observability*. PhD thesis, Uppsala University, 2004.

[39] Davide Falessi, Massimiliano Di Penta, Gerardo Canfora, and Giovanni Cantone. Estimating the number of remaining links in traceability recovery. *Empirical Software Engineering*, 22(3):996–1027, 2017.

[40] Flávio Fernandes and Mark Song. UML-Checker: An Approach for Verifying UML Behavioral Diagrams. *Journal of Software*, 9(5):1229–1236, 2014.

[41] Stephan Flake and Wolfgang Mueller. A UML profille for Real-Time Constraints with the OCL. In *International Conference on the Unified Modeling Language, Berlin, Heidelberg*, pages 179–195. Springer, 2002.

[42] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In *International Conference on Formal Engineering Methods*, pages 581–596. Springer, 2010.

[43] Lukasz Fryz and Leszek Kotulski. Assurance of system consistency during independent creation of UML diagrams. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX'07. 2nd International Conference on*, pages 51–58. IEEE, 2007.

[44] Maroua Gasmi, Olfa Mosbahi, Mohamed Khalgui, and Luis Gomes. Reconfigurable priority ceiling protocol under rate monotonic based real-time scheduling. In *Informatics in Control, Automation and Robotics (ICINCO), 2014 11th International Conference on*, volume 1, pages 42–52. IEEE, 2014.

[45] Abdelouahed Gherbi and Ferhat Khendek. UML Profiles for Real-Time Systems and their Applications. *Journal of Object Technology*, 5(4):149–169, 2006.

[46] Abdelouahed Gherbi and Ferhat Khendek. Consistency of UML/SPT models. *Lecture Notes in Computer Science*, 4745:203, 2007.

[47] Orlena CZ Gotel and CW Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101. IEEE, 1994.

[48] Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations with UML. In *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS 2003), a satellite event of UML*, Oct 2003.

[49] Zonghua Gu and Kang G Shin. Synthesis of real-time implementation from UML-RT models. In *Proceedings of the 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES 04)*, pages 25–28, May 2004.

[50] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne. Improving requirements tracing via information retrieval. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 138–147. IEEE, 2003.

[51] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[52] A Hubaux, A Cleve, PY Schobbens, A Keller, O Muliawan, S Castro, K Mens, D Deridder, and RVD Straeten. Towards a Unifying Conceptual Framework for Inconsistency Management Approaches. Technical report, Technical Report, University of Namur Rue Grandgagnage, 2009.

[53] IBM. Rational Unified Process. http://www-306.ibm.com/software/rational.

[54] INCOSE. http://www.incose.org.

[55] Muhammad Zohaib Iqbal, Shaukat Ali, Tao Yue, and Lionel Briand. Applying UML/MARTE on industrial projects: challenges, experiences, and guidelines. *Software & Systems Modeling*, 14(4):1367–1385, 2015.

[56] ITU-T. *SDL combined with UML*. ITU-T recommandation Z.109, 2000.

[57] Ivar Jacobson. *Object-oriented software engineering: a use case driven approach*. Pearson Education India, 1993.

[58] Ananya Kanjilal, Goutam Kanjilal, and Swapan Bhattacharya. Metrics-based Analysis of Requirements for Object-Oriented systems: An empirical approach. *INFOCOMP Journal of Computer Science*, 7(2):26–36, 2008.

[59] Dániel Kristóf Kiss. Intelligent priority ceiling protocol for scheduling. In *Logistics and Industrial Informatics (LINDI), 2011 3rd IEEE International Symposium on*, pages 105–110. IEEE, 2011.

[60] J Kuster and Joachim Stroop. Consistent design of embedded real-time systems with UML-RT. In *Object-Oriented Real-Time Distributed Computing, 2001. ISORC-2001. Proceedings. Fourth IEEE International Symposium on*, pages 31–40. IEEE, 2001.

[61] Jochen M Küster and Jan Stehr. Towards explicit behavioral consistency concepts in the UML. In *Proceedings of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, 2003.

[62] Kwok-Wa Lam, Sang H Son, Sheung-Lun Hung, and Zhiwei Wang. Scheduling transactions with stringent real-time constraints. *Information Systems*, 25(6-7):431–452, 2000.

[63] Kwok-wa Lam, Sang Hyuk Son, and Sheung-lun Hung. A priority ceiling protocol with dynamic adjustment of serialization order. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 552–561. IEEE, 1997.

[64] Luigi Lavazza, Gabriele Quaroni, and Matteo Venturelli. Combining UML and formal notations for modelling real-time systems. *ACM SIGSOFT Software Engineering Notes*, 26(5):196–206, 2001.

[65] Hong Anh Le, Thi-Huong Dao, and Ninh-Thuan Truong. A Formal Approach to Checking Consistency in Software Refactoring. *Mobile Networks and Applications*, 22(2):356–366, 2017.

[66] Mengjun Li. Automatic proving or disproving equality loop invariants based on finite difference techniques. *Information Processing Letters*, 115(4):468–474, 2015.

[67] Xiaoshan Li, Zhiming Liu, and He Jifeng. A formal semantics of UML sequence diagram. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 168–177. IEEE, 2004.

[68] Xuandong Li and Johan Lilius. Timing analysis of UML sequence diagrams. In *International Conference on the Unified Modeling Language*, pages 661–674. Springer, 1999.

[69] Boris Litvak, Shmuel Tyszberowicz, and Amiram Yehudai. Behavioral consistency validation of UML diagrams. In *Software Engineering and Formal Methods, 2003. Proceedings. First International Conference on*, pages 118–125. IEEE, 2003.

[70] Jane W. S. Liu. *Real-Time Systems.* Prentice Hall, 2000.

[71] Francisco J Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631–1645, 2009.

[72] Philipp Lucas. Timed semantics of message sequence charts based on timed automata. *Electronic Notes in Theoretical Computer Science*, 65(6):160–179, 2002.

[73] Patrick Mader and Alexander Egyed. Assessing the effect of requirements traceability for software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 171–180. IEEE, 2012.

[74] Patrick Mäder and Alexander Egyed. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, 20(2):413–441, 2015.

[75] Milena Rota Sena Marques, Eliane Siegert, and Lisane Brisolara. Integrating UML, MARTE and SysML to improve requirements specification and traceability in the embedded domain. In *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*, pages 176–181. IEEE, 2014.

[76] G. Martin and W. Muller. *UML for SoC Design.* Springer, 2005.

[77] Aloysius Ka-Lau Mok. *Fundamental design problems of distributed systems for the hard-real-time environment.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983.

[78] B Moller-Pedersen. SDL combined with UML. *Telektronikk*, 96(4):36–53, 2000.

[79] Edjard Mota, Edmund Clarke, Alex Groce, Waleska Oliveira, Marcia Falcao, and Jorge Kanda. VeriAgent: an approach to integrating UML and formal verification tools. *Electronic Notes in Theoretical Computer Science*, 95:111–129, 2004.

[80] Wolfgang Mueller, Alberto Rosti, Sara Bocchio, Elvinia Riccobene, Patrizia Scandurra, Wim Dehaene, and Yves Vanderperren. UML for ESL design-basic principles, tools, and applications. In *Computer-Aided Design, 2006. ICCAD'06. IEEE/ACM International Conference on*, pages 73–80. IEEE, 2006.

[81] Elena Navarro, Patricio Letelier, Jose Antonio Mocholi, and Isidro Ramos. A metamodeling approach for requirements specification. *Journal of Computer Information Systems*, 46(5):67–77, 2006.

[82] Ariadi Nugroho and Michel RV Chaudron. A survey into the rigor of UML use and its perceived impact on quality and productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 90–99. ACM, 2008.

[83] OMG. *UML Superstructure Specification.* version 2.0, 2004.

[84] OMG. *UML Superstructure Specification.* version 2.3, 2010.

[85] OMG. *UML Profile for Schedulability, Performance and Time Specification.* omg document formal/05-01-02 edition, Jan 2005.

[86] OMG. *Meta Object Facility (MOF) Core Specification.* OMG Available Specification version 2.0 formal/06-01-01, Jan 2006.

[87] OMG. *UML 2.0 Superstructure Specification.* omg document formal/05-07-04 edition, Jul 2005.

[88] OMG. *UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms.* OMG Adopted Specification, ptc/2004-06-01, Jun 2004.

[89] OMG. *UML Profile for System on a Chip (SOC).* Available Specification Version 1.0 formal/06-06-01, Jun 2006.

[90] OMG. *UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems.* OMG document formal/2011-06-02 edition, Jun 2011.

[91] OMG. *Unified Modeling Language Specification.* OMG document formal/03-03-01 edition, Mar 2003.

[92] OMG. *Object Constraint Language (OCL).* OMG Available Specification version 2.0 formal/06-05-01, May 2006.

[93] OMG. *Systems Modeling Language (OMG SysML) Specification.* Final Adopted Specification ptc/06-05-04, May 2006.

[94] OMG. *UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems.* OMG document formal/2009-11-02/ edition, Nov 2009.

[95] T Parr. The Definitive ANTLR Reference-Building Domain-Specific Languages. Pragmatic Bookshelf. Technical report, ISBN 978-0-9787392-5-6, 2007.

[96] Marie-Agnès Peraldi-Frati and Arnaud Albinet. Requirement traceability in safety critical systems. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, pages 11–14. ACM, 2010.

[97] Carl Adam Petri. Kommunikation mit automaten. 1962.

[98] Francisco AC Pinheiro. Formal and informal aspects of requirements tracing. In *WER*, pages 1–21, 2000.

[99] Francisco AC Pinheiro. Requirements traceability. In *Perspectives on software requirements*, pages 91–113. Springer, 2004.

[100] Ernesto Posse and Juergen Dingel. An executable formal semantics for UML-RT. *Software & Systems Modeling*, 15(1):179–217, 2016.

[101] R. Rajkumar. *Synchronization in real-time systems: a priority inheritance approach*. Springer Science & Business Media, 2012.

[102] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[103] Patrick Rempel and Parick Mäder. Preventing defects: the impact of requirements traceability completeness on software quality. *IEEE Transactions on Software Engineering*, 43(8):777–797, 2017.

[104] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingolfsdottir, and Steinar Hugi Sigurdarson. Modelling and simulation of asynchronous real-time systems using Timed Rebeca. *Science of Computer Programming*, 89:41–68, 2014.

[105] Fabíola Gonçalves C Ribeiro, Carlos E Pereira, Achim Rettberg, and Michel S Soares. Model-based requirements specification of real-time systems with UML, SysML and MARTE. *Software & Systems Modeling*, 17(1):1–19, 2016.

[106] Doug Rosenberg and Matt Stephens. Use case driven object modeling with UML. *APress, Berkeley, USA*, 2007.

[107] RTL. Software and Tools for Communicating Systems. http://www.laas.fr/RTLOTOS.

[108] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991.

[109] Zeynab Sabahi-Kaviani, Ramtin Khosravi, Peter Csaba Ölveczky, Ehsan Khamespanah, and Marjan Sirjani. Formal semantics and efficient analysis of Timed Rebeca in Real-Time Maude. *Science of Computer Programming*, 113:85–118, 2015.

[110] Mehrdad Sabetzadeh, Shiva Nejati, Marsha Chechik, and Steve Easterbrook. Reasoning about consistency in model merging. *Proc. of Living with Inconsistency in Software Development*, 10, Sep 2010.

[111] Mehrdad Sabetzadeh, Shiva Nejati, Sotirios Liaskos, Steve Easterbrook, and Marsha Chechik. Consistency checking of conceptual models via model merging. In *Requirements Engineering Conference, 2007. RE 07. 15th IEEE International*, pages 221–230. IEEE, Oct 2007.

[112] Sadia Sahar, Tasleem Mustafa, Farnaz Usman, Aasma Khalid, Nadia Aslam, and Sidra Hafeez. Description of Consistency Rules in Software Design Models. *Journal of Emerging Trends in Computing and Information Sciences*, 5(5):428–432, 2014.

[113] Hossein Saiedian and S Raghuraman. Using UML-based rate monotonic analysis to predict schedulability. *IEEE Computer*, 37(10):56–63, 2004.

[114] Manas Saksena and Panagiota Karvelas. Designing for schedulability: integrating schedulability analysis with object-oriented design. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 101–108. IEEE, 2000.

[115] PG Sapna and Hrushikesha Mohanty. Ensuring consistency in relational repository of UML models. In *information Technology,(ICIT 2007). 10th international conference on*, pages 217–222. IEEE, 2007.

[116] Bran Selic. Using UML for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*, pages 250–260. Springer, 1998.

[117] Bran Selic and Sebastien Gerard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE.* Elsevier, 2014.

[118] Bran Selic, Garth Gullekson, and Paul Ward. Real-time object oriented modeling and design. *J. Wiley & Sons*, 1994.

[119] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[120] Yoshiyuki Shinkawa. Inter-model consistency in UML based on CPN formalism. In *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pages 411–418. IEEE, 2006.

[121] Marwa Shousha, Lionel Briand, and Yvan Labiche. A UML/MARTE model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. *IEEE Transactions on Software Engineering*, 38(2):354–374, 2012.

[122] George Spanoudakis. Plausible and adaptive requirement traceability structures. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 135–142. ACM, 2002.

[123] George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. *Handbook of software engineering and knowledge engineering*, 1:329–380, 2001.

[124] George Spanoudakis, Andrea Zisman, Elena Pérez-Minana, and Paul Krause. Rule-based generation of requirements traceability relations. *Journal of systems and software*, 72(2):105–127, 2004.

[125] OMEGA ST. Project. http://www.omega.imag.fr.

[126] Laura Florentina Stoica and Florian Mircea Boian. Algebraic approach to implementing an ATL model checker. *Studia Univ. Babes Bolyai, Informatica, Cluj-Napoca, Romania*, 57(2):73–82, 2012.

[127] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):3.1–3.29, 2013.

[128] SystemC. The Open SystemC Initiative. http://www.systemc.org.

[129] Telelogic. Telelogic Products. http://www.telelogic.com/products.

[130] TTool. A Toolkit for Editing and Validating TURTLE Diagrams. http://www.eurecom.fr/ apvrille/TURTLE/index.html.

[131] Maria Cruz Valiente, Gonzalo Genova, and Jesus Carretero. UML 2.0 Notation for Modeling Real Time Task Scheduling. *Journal of Object technology*, 5(4):91–105, 2006.

[132] Mandana Vaziri and Daniel Jackson. Some shortcomings of OCL, the object constraint language of UML. In *TOOLS (34)*, pages 555–562, 2000.

[133] Xibo Wang, Fenmei Wang, and Ge Yu. Research on Resource Access Control Protocol Based on Layered Scheduling Algorithm. In *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*, pages 134–139. IEEE, 2008.

[134] Stefan Winkler and Jens Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling (SoSyM)*, 9(4):529–565, 2010.

[135] Fengxiang Zhang and Alan Burns. Schedulability analysis of EDF-scheduled embedded real-time systems with resource sharing. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(3):67.1–67.18, 2013.

[136] Xingyuan Zhang, Christian Urban, and Chunhan Wu. Priority inheritance protocol proved correct. *Interactive Theorem Proving*, pages 217–232, 2012.

[137] Yonggang Zhang, René Witte, Juergen Rilling, and Volker Haarslev. Ontological approach for the semantic recovery of traceability links between software artefacts. *IET software*, 2(3):185–203, 2008.

[138] Xiangpeng Zhao, Quan Long, and Zongyan Qiu. Model checking dynamic UML consistency. *Formal methods and software engineering*, pages 440–459, 2006.

[139] Gregory Zoughbi, Lionel Briand, and Yvan Labiche. Modeling safety and airworthiness (RTCA DO-178B) information: conceptual model and UML profile. *Software and Systems Modeling*, 10(3):337–367, 2011.