JAIST Repository
https://dspace.jaist.ac.jp/

Title	車載ネットワークシステムのモデル検査に関する研究
Author(s)	郭，暁芸
Citation	
Issue Date	2019-03
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/15793
Rights	
Description	Supervisor:青木　利晃，情報科学研究科，博士

JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Model Checking of In-vehicle Networking Systems

Xiaoyun Guo

Japan Advanced Institute of Science and Technology

Doctoral Dissertation

Model Checking of In-vehicle Networking Systems

Xiaoyun Guo

Supervisor: Toshiaki Aoki

School of Information Science
Japan Advanced Institute of Science and Technology

March, 2019

Abstract

In-vehicle networking (IVN) systems consist of electronic components that are connected by buses and communicate through multiple protocols according to their requirements. Communications between these subsystems are getting more complicated as the requirements for safety, comfort, and entertainment. Different communication protocols have their special mechanisms to transmit messages on buses, which affect safety and timed property of the IVN system. In practice, intelligent vehicles need to exchange safety data between subsystems that use various protocols, such as the Controller Area Network (CAN) and FlexRay. Such systems are more likely to encounter delays and message loss during transmission, presenting serious safety issues. Moreover, IVN systems are extremely complicated because of their large number of nodes, multiple communication protocols, and diverse topologies. As a result, it is difficult to check properties of the system directly and accurately. Besides, safety-critical events occur with a probability in the IVN system, such as the probability of failures and the probability of emergency events happened during driving. These probabilistic events are crucial to estimate real-time and reliability of the IVN system.

In this work, we propose a framework based on UPPAAL model checker, for modeling and verifying communicative behaviors between multiple protocols in an IVN system. Due to the complicated of the IVN system, we present an appropriate abstraction with two stages for modeling IVN systems that utilize CAN and FlexRay during the design phase. The architecture of the IVN system is abstracted to reduce the number of nodes first, and then the composition of each protocol is abstracted to simplify states of systems based on protocol specifications. As there are numerous IVN system structures, a reusable framework is developed to build a design model for IVN systems with different topologies. In the framework, an IVN system model consists of protocol, interface, configuration, forwarder and environments modules. The environments and forwarder modules are changeable according to system design. The protocol, interface and configuration modules are fixed to construct various IVN systems.

Using this framework, we check IVN systems from qualitative verification and quantitative verification. Through the qualitative verification, the timed properties of communication are analyzed using the UPPAAL platform; we verify the reachability and response time of messages in the best case and worst case. Through the quantitative verification, the probability of message reachability during a time interval is given by application probability models in the SMC-UPPAAL; the probability density and probability distribution of response time is used to analyze the frequency for receiving messages. The two verifications complement each other. The qualitative verification is exhaustive, but the efficiency and capability is limited. The quantitative verification is more efficient, but the properties are satisfied with some degree of confidence.

The framework is evaluated through several aspects. First, we evaluate the validity of the abstraction. The framework is preservation for outside of the subsystem, however, the inside of subsystem can not be preserved. We list properties from specifications, and

the framework is validated by checking the communication behavior against the protocol specifications and some properties can be checked. But some properties cannot be checked because of the abstraction. Second, we demonstrate the applicability of the framework with three typical topologies. Third, we compare source code of three different systems in UPPAAL, the framework is reusable for different system with little change. Finally, we show the performance of the framework in qualitative verification and quantitative verification.

Keywords: Model Checking, Statistical Model Checking, UPPAAL, In-vehicle Network System, CAN and FlexRay.

Acknowledgments

This research has been conducted under School of Information Science, Japan Advanced Institute Science and Technology (JAIST). This research could not be completed without supports and helps from numerous people.

First of all, I would like to express my sincere gratitude to my supervisor, Prof. Toshiaki Aoki, who supervised me since I started my master's program. He gave me great help on my research. He taught me how to do research, and pointed out many problems and gave me a lot of valuable comments and guiding suggestions. Also, he helped me a lot in living.

Besides, I would like to express my honest appreciation to Dr. Hsin-hung Lin, who was my senior in Aoki Laboratory. He always discussed my research and problems with me. I also would like to thank Dr. Yuki Chiba sincerely, who was a Assistant Professor in our laboratory. He taught me many basic knowledge about mathematical logic and automata theory, and gave me comments on my research.

In addition, I wish to gratitude to all committee members including of: Professor Yasuo Tan, Professor Kunihiko Hiraishi, Associate Professor Masato Suzuki, and Professor Kozo Okano. They gave me many useful comments, so that I could have improved my research.

I would like to thank to my colleagues and friends for their kindness. Their help and companionship made me have a good time in Japan.

Last but not least, I would like to express my deepest gratitude to my parents for their support and encouragement all the time. Without their great love and understanding, I could not have finished this dissertation successfully.

Contents

Abstract i

Acknowledgments iii

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Objective and Approach . . . 5
- 1.3 Contribution . . . 7
- 1.4 Outline of This Thesis . . . 8

2 Preliminaries 9
- 2.1 Model Checking . . . 9
- 2.2 Timed Automata . . . 13
- 2.3 UPPAAL . . . 14
  - 2.3.1 Modeling Language . . . 15
  - 2.3.2 Specification Language . . . 16
- 2.4 UPPAAL-SMC . . . 17
  - 2.4.1 Modeling Language . . . 17
  - 2.4.2 Specification Language . . . 17
- 2.5 In-vehicle Networking System . . . 18
  - 2.5.1 Controller Area Network . . . 19
  - 2.5.2 FlexRay . . . 20
  - 2.5.3 Topologies of IVN systems . . . 22

3 Abstractions of IVN Systems 24
- 3.1 Abstracting the Architecture of IVN Systems . . . 24
- 3.2 Abstracting the Composition of Protocol Specifications . . . 27

3.2.1 Overview of Protocol Specifications 28
3.2.2 The Composition of protocol specifications 31
3.3 Communication Control Models 33
3.3.1 CAN Communication Control Model 34
3.3.2 FlexRay Communication Control Model 36
3.3.3 Media Access Control 39
3.3.4 Frame and Symbol Processing 49

4 A Framework for Modeling IVN Systems 52
4.1 A Framework in UPPAAL 52
4.2 Configuration Module 61
4.3 Interface and Medium Module 63
4.4 Environmnet Module 65
4.4.1 Ordinary Environment Model 66
4.4.2 Probabilistic Environment Model 66
4.5 Forwarder Module 70
4.6 An IVN System Design Model 72
4.7 Message Transmission 73

5 Verification and Evaluation 81
5.1 Qualitative Verification 81
5.1.1 Validity of the IVN System Model 81
5.1.2 Reachability 85
5.1.3 Response Time 85
5.2 Quantitative Verification 87
5.2.1 Application Model with Probability 87
5.2.2 Reachability and Response Time 89
5.3 Evaluation 92
5.3.1 Validity of the Abstraction 93
5.3.2 Applicability 96
5.3.3 Reusability 100
5.3.4 Performance 104

6 Related Work 106
6.1 Verification of IVN Systems Based on Integration Platforms 106
6.2 Model Checking of IVN Systems 107
6.3 Statistical Model Checking of IVN Systems 108

7 Conclusion and Future Work 109
7.1 Conclusion 109
7.2 Future Work 112

List of Figures

1.1 Message response times. 4

2.1 Model checking method 10
2.2 Example of CAN message transmission. 20
2.3 Example of FlexRay message transmission. 21
2.4 Three topologies of IVN systems. 22

3.1 Subnetwork topologies 26
3.2 Abstraction for the architecture of IVN systems. 28
3.3 Layered structures of the CAN protocol and FlexRay protocol. 29
3.4 Architecture of the FlexRay protocol. 30
3.5 The common structure of the CAN specification and FlexRay specification. 32
3.6 Structure of the CAN communication controller. 34
3.7 Arbitration automaton. 35
3.8 Transceiver automaton. 35
3.9 Structure of the FlexRay communication controller. 36
3.10 Overview of protocol operation control. 37
3.11 POC automaton. 38
3.12 Communication cycle of FlexRay model 39
3.13 Media access process 41
3.14 Media access in static segment 42
3.15 StaticMAC automaton. 44
3.16 Media access in dynamic segment 45
3.17 (a) action point a (b) action point b 46
3.18 Media access in dynamic segment arbitration 47
3.19 Dynamic MAC automaton 48
3.20 Architecture of the IVN system design. 49
3.21 NIT automaton 49
3.22 Overview of frame and symbol processing 50
3.23 FSP automaton 50

4.1 The framework. 53
4.2 The class diagram of the framework in UML. 54
4.3 The component diagram. 59

4.4 The hierarchy diagram of the *Configuration*. 62
4.5 Environment automata for writing messages. 67
4.6 Environment automaton for reading messages. 67
4.7 Environment automaton with probabilistic choices. 68
4.8 Distribution of reachability time with probabilistic choices. 68
4.9 Environment automaton with uniform distributions. 69
4.10 Uniform distribution of reachability time. 69
4.11 Environment automaton with exponential distributions. 69
4.12 Exponential distribution of reachability time. 70
4.13 GatewayController fragment for monitoring the *Interface*. 71
4.14 *ForwardController* fragment for forwarding messages. 71
4.15 A *ForwardController* automaton. 76
4.16 Architecture of the IVN system design. 77
4.17 Environment model for the IVN system design. 77
4.18 Six possible system topologies. 78
4.19 The sequence diagram of the Case 1. 78
4.20 The sequence diagram of the Case 2. 79
4.21 The sequence diagram of the Case 3. 79
4.22 The sequence diagram of the Case 4. 80
4.23 The sequence diagram of the Case 5. 80
4.24 The sequence diagram of the Case 6. 80

5.1 Observer for checking response time. 85
5.2 Observer for checking response time. 86
5.3 Transceiver automaton for statistical model checking. 88
5.4 Tasks automata and distributions of reachability time in Design 1. 89
5.5 Probability density distribution of response time in Design 1. 91
5.6 Tasks automata and distributions of reachability time in Design 2. 92
5.7 Probability density distribution of response time in Design 2. 93
5.8 Restored architectures of a subsystem. 97
5.9 Topologies in *Case1*. 98
5.10 Task automaton for each E1 of the three IVN systems in *Case1*. 98
5.11 Topologies in *Case2* and *Case3*. 99
5.12 Task automaton for each E1 of the three IVN systems in *Case2*. 99
5.13 Task automata for each E1 and E2 of the three IVN systems in *Case3*. . . 100
5.14 The topologies of three systems. 101

List of Tables

2.1 CAN and FlexRay comparison 19

4.1 Parameters in the *Configuration* 58
4.2 Configuration of the IVN system design 74

5.1 Comparison of models between model checking and statistical model checking 88
5.2 Properties in CAN specifications 94
5.3 Properties in FlexRay specifications 95
5.4 Comparison of response times of the `FRMsg` in different topologies. 97
5.5 Response time of each topology in the three cases. 100
5.6 Comparison of source code of the three systems. 102
5.7 Performance of qualitative verification 105
5.8 Performance of quantitative verification 105

Chapter 1

Introduction

1.1 Background and Motivation

Modern automobile industry has replaced most traditional mechanical and hydraulic system with X-by-Wire technique. The X-by-Wire technique comes from Fly-by-Wire on the airplane, where "W" refers to a specific electronic control system, such as Break-by-Wire and Steer-by-Wire. In such systems, they utilize electronic control units (ECUs) as controllers instead of mechanical components. These controllers process data detected from sensors and transmit results (electronic control signals) to actuators, which perform control operations. Between the sensors, controllers and actuators, the data are transmitted by buses. The X-by-Wire technique greatly simplifies the structure of the vehicle and improves the real time and flexibility. To meet security, comfort and entertainment requirements, electronic control systems become more and more diverse in an vehicle, and they constitute an in-vehicle networking (IVN) system. The IVN system is a real-time distributed system consisting of multiple subsystems, each of which employs different communication protocols to realize data transmission according to its requirements. The controller area network (CAN), local interconnect network (LIN), FlexRay, media oriented systems transport (MOST) and other protocols are applied in IVN systems. CAN is a field-bus with a wide range of application areas, which has good fault tolerant and is often used in powertrain system and body control system of the vehicle. FlexRay is a high-speed, fixable and reliable communication protocol aimed at IVN systems. It is applicable to control systems with high requirements on real-time, security and reliability. LIN is a simple low-speed bus as a complement to other buses. It is suitable for controlling seats, air-condition, lights and so on. MOST was developed for the in-car multimedia system with high bandwidth, and is used in navigation, television, CD player and other

entertainment systems. These subsystems, which use different protocols, connect to a central gateway to transfer data. Since the IVN system is a complex multi-protocol communication system, which is universally used in practical cars, and directly related to life security, it is extremely important to verify the reliability and safety of the IVN system.

There are two main aspects to check the IVN system, software (operation systems) and hardware (communication protocols). The software and hardware are interdependent and interactional, and have to satisfy their standards, such as software standard: AUTOSAR and OSEK/VDX, hardware standard: CAN and FlexRay. In automotive industry, using an integrated platform is the most common test method, combining software and hardware. For example, DaimlerChrysler laboratory developed a tool including software and hardware architectures to flexibly construct and test IVN systems [1]; T. Demmeler et al. proposed a virtual integration platform to simulate and estimate the performance of IVN communication models [2]; H.Moon et al. implemented a heating ventilation and air-condition control system based on AUTOSAR architecture and tested it using MATLAB and SIMULINK [3]. There are other studies that are directly implemented an IVN system on integrated components and simulate and test electronic signal on hardware. For example, G. Feng et al. implemented a CAN system with electronic nodes and tested the system [4]; F. Baronti et al. had designed and implemented a FlexRay protocol component to verify fault tolerance of IVN systems [5]. Some studies only test and analyze the system itself. For instance, S. Anssi et al. focused on analyzing scheduling capability of AUTOSAR system [6]. Although these studies can be used to analyze and test IVN systems concretely and intuitively, a completed set of test cases is difficult to design for checking properties of the system. Also, it is hard to precisely check concurrent behaviors and logic errors in the system design phase.

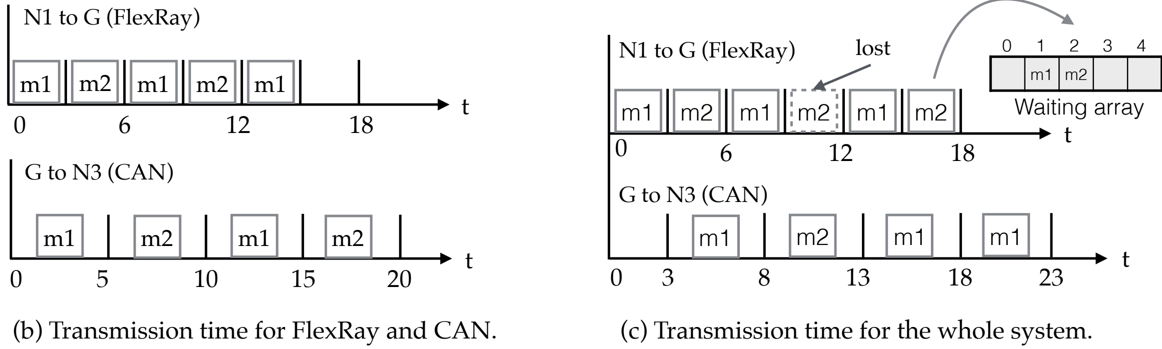
Model checking is another effective method on verification of safety-critical systems. This method is exhaustively and automatically to check properties by searching all states of the system. The properties can be specified in temporal logic and verified precisely. There are many works on checking communication protocols [7, 8, 9]. They modeled protocols and verified properties from their specifications, such as the fault-tolerance on the FlexRay physical layer, the start-up process of FlexRay, and the error handling and fault-tolerance of timed-triggered CAN. For the software, some works [10, 11, 12, 13] aimed at verifying that implementations of the system is consistent with software standards. J. Chen et al. designed a model based on OSEK/VDX operating system and verified that the model meets OSEK specification by SPIN, and generated test cases using the model [11]. L. Fang et al. proposed a formal model of AUTOSAR multicore real-time operating system and developed a test case generator and a test program generator to complete

system testing [13]. Y. Huang et al. implemented a formal model of OSEK/VDX OS with CSP language and verified the model using PAT [12]. In addition, some studies focus on interactive behaviors between ECUs in a single protocol [14, 15]. L.Waszniowski et al. established a whole CAN system model with OSEK/VDK operating system and showed a case study to verify timed properties of the system [14]. C.Pan et al. showed a CAN protocol model to verify some primary properties using UPPAAL model checker, and considered an application model with a scheduling algorithm to fix some properties that are unsatisfied [15]. However, an IVN system is extremely complicated, which contains multiple protocols, several gateways, and many applications. It is difficult to directly model the complete IVN system and efficiently verify properties. Most of researches take into account one part of the IVN system, and few studies can consider a complete IVN system architecture with multiple protocols. Moreover, the capability of verification is usually limited because the complexity of the system cause the state space explosion.

Communication among multiple protocols is inevitable and the amount of data will be more and more, specially in an intelligent vehicle. For instance, driver assistance systems and automated driving systems need to monitor the vehicle situation in real time and deal with events by controlling the corresponding node in time. Such systems capture and analyze data from each subsystem that may use different communication protocols, such as between the body control subsystem with CAN and the chassis control subsystem with FlexRay [5]. Usually, every protocol has special communication mechanism and frame format, and we check properties of each subsystem separately, for example [14, 15] mentioned earlier. However, the IVN system is a whole in which subsystems with different protocol may influence each other, the checking results of the whole IVN system may differ from those separately checked.

There is a simple communication system with CAN and FlexRay for clarifying above problem. This system consists of three nodes, N1 follows FlexRay protocol to transmit messages, N2 follows CAN protocol to transmit messages and G is a gateway connecting N1and N2. The topology of the system is shown in Fig. 1.1 (a). CAN protocol specifies that messages are transmitted to CAN bus in turn according to a static priority algorithm, and FlexRay protocol specifies that messages are transmitted to fixed time slots of a communication cycle, or like CAN protocol based on the message priority. N1 repeatedly sends two messages, `m1` and `m2`, to G, where the `m1` has higher priority than the `m2`, and these messages will be forwarded to N2 through CAN bus. The communication cycle of FlexRay has two slots, `m1` is assigned to slot 1 and `m2` is assigned to slot 2, and each slot is 3 time units. Since the transmission rate of CAN is slower than FlexRay, we assume that it takes 5 time units to transmit `m1` and `m2`. In real-time systems, it is important to verify

(a) The structure of the system.



(b) Transmission time for FlexRay and CAN.

(c) Transmission time for the whole system.

Figure 1.1: Message response times.

timed properties such as the message transmission time satisfying a time constraint. If we examine FlexRay part and CAN part of the system separately, we will get the transmission time of the messages shown in Fig. 1.1 (b). In the FlexRay part, `m1` and `m2` are sent in the fixed time slots and they alternately reaches G node. The transmission times of these messages are same, 3 time units. In the CAN part, referring to the transmission result of the FlexRay part, `m1` and `m2` are transmitted in turn and the transmission times of these messages are 5 time units. As for the message transmission times in the whole system, we can combine the two results and it is 8 time units.

When we observe the whole system in a time line, the transmission time of the messages is shown in Fig. 1.1 (c). As the messages are sent from N1 to N2, the transmission time from N1 to G under FlexRay will affect the transmission time from G to N2 by CAN. From N1 to G, the transmission time is same as Fig. 1.1 (b). After the message reached G node, there is a waiting array to store received messages in the G node, and the G node will forward messages to CAN network. When the first `m1` is received and stored in the array in accordance with its priority, the time is 3. Instantly, G sends the message to CAN bus, and it will be received at `t=8`. While `t=8`, the first `m2` was already waiting in the array. Hence, the `m2` is transmitted immediately, and at `t=13`, the message will be obtained by N2. At this moment, the FlexRay part of the system has finished the second communication cycle, and the second `m1` and `m2` existed in the array. Since the priority of `m1` is the highest one, the G node sends the second `m1` to N2 first. But when the transmission of the message ends in CAN, that is `t=18`, the third communication cycle also ends. The second `m2` that was waiting to be sent has been overwritten by

the new message. In the whole system, the G node periodically receives the `m1` and `m2` according to the 6-time-unit communication cycle of FlexRay. All `m1` can be successfully sent to N2 because of the higher priority, but some `m2` that wait more than 6-time-unit in the array will be lost. Additionally, the transmission time of every `m1` is different, for example, the first `m1` took 8-time-unit and the second `m1` took 12-time-unit from N1 to N2. If there is a timed property that demands the transmission time for all messages is less than 10-time-unit, the property is unsatisfied in considering multiple protocols. As a result, the checking result of the timed property of the IVN system will be affected by different communication mechanisms. Hence, the timed properties of the communication between multiple protocols should be taken into account.

Furthermore, there are safety-critical events that happen with a probability in the IVN system and the probabilities are calculated by some studies [16, 17]. The probability of events is utilized to check reliability of the system. For instance, [18] analyzes the dependability of safety-relevant systems using failure mode and effects analysis. These probabilistic events are crucial to estimate real-time of the IVN system. Nevertheless, classical model checking method does qualitative verification of properties, and such probabilistic behaviors can not described. Statistical model checking does quantitative verification, and describe behaviors with probabilities, but it does not guarantee precise results.

Thus, this research was stimulated to propose a framework for modeling and verifying IVN systems during the design phases. We concentrate on the transmission time of messages between CAN and FlexRay subsystems, as these are widely used in the automotive industry and check timed properties from both the quality and quantity using model checking techniques. To analyze the timed property, we describe the modeling and verification of IVN system design models using the UPPAAL platform.

1.2 Objective and Approach

The objective of this work is to provide an approach to verifying the timed property of the IVN system in the presence of CAN and FlexRay in the design phase. There are three challenges to achieving the objective: 1) it is vital to find a proper abstraction for modeling IVN systems; 2) a reusable framework is required to model the various topologies that arise from multiple protocols; 3) the IVN system design model should be checked by qualitative verification and quantitative verification. Afterwards, we analyze and propose solutions to each difficulty.

- A real IVN system needs to meet both software and hardware standards, such as communication protocols and operating system standards we mentioned in the last

section. These standards are varied, and explain in detail with regard to application, scheduling mode, hardware implementation, communication mechanism and so on. Modeling every detail of such a complex system would make it impossible to verify the functionality because of the state space explosion problem. Thence, it is necessary to abstract the IVN system to make the state space as small as possible. However, a model that is too abstract may not be able to verify the timed properties of the system accurately. For this reason, there must be a trade-off between accuracy and abstraction. Although several studies have discussed the modeling of the CAN and FlexRay protocols separately [10, 9], there has been little research on IVN systems operating both. System in which different kinds of nodes co-exist will operate under multiple protocols, which make the abstraction non-trivial. Thus, to overcome this issue, we take up a two-staged strategy to derive abstractions for both the architecture and function of the system.

- IVN systems may have several subsystems with manifold applications and topologies by designer. These subsystems are connected together via gateways and communicate with each others. In the subsystem, not all nodes and data participate the communication with other subsystems. Hence, we abstract a subsystem as an environment node, ignoring its topology and internal communication, to simplify the system architecture.
- Nodes in the IVN system are specified by communication protocols and OS standards respectively. The CAN and FlexRay protocol specifications contain a large of details about implementations on hardware, which are not connected verification of design models of IVN systems. Besides, we assume that communication in the IVN system is error-free. Therefore, when constructing the system model, abstractions are applied to remove the low level behaviors as well as functions related to hardwares not needed for verification of design models. The layer structure of the nodes is in accordance with the ISO model, where the application layer is described by the OS standards. This layer is used to implement data processing and control functions of IVN systems designed by users. Since we have abstracted multiple nodes in the subsystem into one, this node performs the task of sending messages to other systems. We ignore task executions in the application layer, and abstract a take to generate messages that need to be transmitted. Based on the layered structure of nodes described in the protocol specifications, we propose an abstracted structure for CAN and FlexRay nodes.

- Because of the diversity of the IVN system, whether it is the topology or application, a model does not seem to fit all situations. Hence, a reusable framework is required to model various topologies that arise from multiple protocols. Based on the similarity of the abstracted node structure, we propose a framework to model IVN systems. The framework is composed of application layer, object layer, transfer layer and configuration based on UPPAAL. The object layer, transfer layer and configuration are reusable for building system design models with varying topologies by modifying application and parameters in the configuration. Using the UPPAAL platform, we implement a design model to evaluate that the framework is validated by checking the communication behavior against the protocol specifications. Using the framework, we establish a series of design models with three typical topologies: central, backbone, and daisy chain. The applicability and reusability of the framework over these topologies was appraised by comparing the checking results of response time and source codes in UPPAAL.

- IVN systems are distributed embedded systems, which usually have stochastically behaviors, such as inputs, message delays and failures [19]. The behaviors of each ECU in the IVN system are independent and may be executed with a probability. If such behaviors of the complex IVN system are only tested for statistical analysis, it is hard to provide an accurate result. Although model checking technique can exhaustively and automatically check the systems with high accuracy, state space problem may limits efficiency and capability for verifying such complex systems. Therefore, we consider statistical model checking to verify the IVN systems with the probability behaviors. Fortunately, UPPAAL model checker supports both model checking and statistical model checking. We construct design models based on the framework in the UPPAAL, and do quantitative verification using statistical model checking through changing little part of the IVN system model. The application module in the framework can be replaced by task model with probability behaviors. The reachability and response time of messages are checked by quantitative verification, and the results are satisfied with a degree of confidence. The probability density distribution of the response time is used to analyze communication efficiency.

1.3 Contribution

This work proposed an approach that verifies communication behaviors of IVN systems with multiple protocols. The two-stage abstraction transforms an extremely complex IVN system into a verifiable model. CAN and FlexRay are real protocols using in the auto-

motive industry. We have modeled the two protocols and verified IVN systems with both CAN and FlexRay based on the abstraction. The framework provides a way to construct IVN system design models, timed model checking to communication, and time behaviors of IVN system designs operating both CAN and FlexRay protocols. The modules of the framework are applicable and reusable so that it is easier to model and verify different IVN system designs with little effort. To the best of our knowledge, this is the first attempt on model checking in which communication behavior of IVN system design with multiple protocols. Moreover, the framework can be reused to check probability behaviors by modifying the application models.

1.4 Outline of This Thesis

The remainder of this thesis is organized as follows. In chapter 2, we discuss the background to this work. The abstraction strategy and the framework are then proposed in chapter 3. Chapter 4 shows how to model an IVN system design using the framework. The qualitative verification and quantitative verification are shown in Chapter 5, and we evaluate the framework. Chapter 6 gives a concise review of related work, before chapter 7 concludes this thesis and outlines ideas for future work.

Chapter 2

Preliminaries

2.1 Model Checking

Model checking [20] is an exhaustive and automatic verification technique for finite state concurrent systems. It has been successfully used in may fields, such as computer hardware/software design, communication protocols, industry, etc. The process of model checking has three steps, modeling, specification, and verification, as shown in Fig. 2.1. Firstly, a design is described abstractly as a formal model (i.e., finite-state transition graph). Then, a specification is given by logical formalisms (i.e., temporal logic), which indicates all properties that the design should satisfy. Finally, the properties are automatically verified, and the results show whether the design model is satisfied. If a property is unsatisfied, a counterexample is provided to help designer for finding errors.

Formally, model checking problem can be stated by $M, s \vDash f$, where M is a Kripke structure, s is all states of M and f is a formula of temporal logic [21]. E. M. Clarke and E. A. Emerson introduced a polynomial algorithm to verify whether the Kripke structure M satisfies a formula of computation tree logic (CTL). The Kripke structure is a type of state-transition graph to represent behaviors of a system, which depicts all states of the system, state transition and conditions of changing states, as defined in the book [20].

Definition 1 (Kripke structure) Let AP be a non-empty set of atomic propositions. A $Kripkestructure$ M over AP is a four tuple $M = (S, S_0, R, L)$, where

- S is a finite set of states;
- $S_0 \in S$ is a set of initial states;
- $R \subseteq S \times S$ is a transition relation that must be hold for every state $s \in S$ there is a

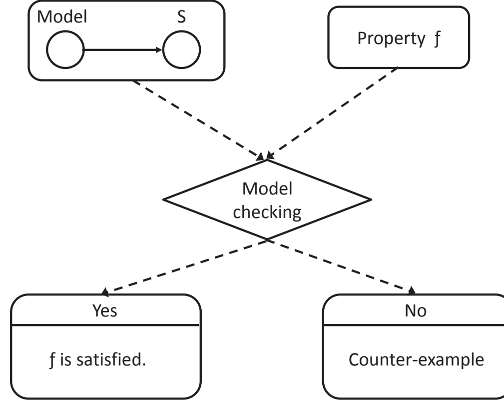


Figure 2.1: Model checking method

state $s' \in S$ such that $R(s, s')$;

- $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

Properties of the Kripke structure are expressed by temporal logic that comprises atomic propositions and boolean connectives. Computation Tree Logic (CTL) is an efficient formalism for describing all of possible transitions between states in a system. A formula (property) that conforms to CTL consists of path quantifiers and temporal operators. The path quantifiers are **A** (for all paths) and **E** (there exists a path). The temporal operators are **X** (next time), **F** (eventually), **G** (always), **U** (until) and **R** (release).

Definition 2 (Syntax)) The syntax of CTL over atomic propositions AP with $p \in AP$ is given below,

$$\psi ::= true|false|p$$

$$|\psi \wedge \psi|\psi \vee \psi|\neg\psi$$

$$|\mathbf{AX}\psi|\mathbf{AF}\psi|\mathbf{AG}\psi|\psi\mathbf{AU}\psi|\psi\mathbf{AR}\psi$$

$$|\mathbf{EX}\psi|\mathbf{EF}\psi|\mathbf{EG}\psi|\psi\mathbf{EU}\psi|\psi\mathbf{ER}\psi$$

Definition 3 (Semantics)) Let M be a Kripke structure, s be a state in M and psi be an CTL formula. Then the satisfaction relation $M, s \models \psi$ is defined as follows.

- $M, s \models true$

- $M, s \vDash false$
- $(M, s \vDash p)$ iff $(p \in L(s))$
- $(M, s \vDash \neg\psi)$ iff $(M, s \nvDash \psi)$
- $(M, s \vDash \psi \wedge \varphi)$ iff $((M, s \vDash \psi)) \wedge (M, s \vDash \varphi))$
- $(M, s \vDash \psi \cup \varphi)$ iff $((M, s \vDash \psi)) \cup (M, s \vDash \varphi))$
- $(M, s \vDash \mathbf{AX}\psi)$ iff $(\forall\pi$ such that $\pi_0 = s, M, \pi^1 \vDash \psi)$, for all paths that start with s, next time ψ can be satisfied.
- $(M, s \vDash \mathbf{AF}\psi)$ iff $(\forall\pi$ such that $\pi_0 = s, \exists i \geqslant 0$ such that $M, \pi^i \vDash \psi)$, for all paths that start with s, eventually ψ can be satisfied.
- $(M, s \vDash \mathbf{AG}\psi)$ iff $(\forall\pi$ such that $\pi_0 = s, \forall i \geqslant 0 M, \pi^i \vDash \psi)$, for all paths that start with s, forever ψ can be satisfied.
- $(M, s \vDash \psi\mathbf{AU}\varphi)$ iff $(\forall\pi$ such that $\pi_0 = s, \exists i \geqslant 0$ such that $(\forall 0 \leq j < i(M, \pi^j \vDash \psi)) \wedge (M, \pi^i \vDash \varphi))$, for all paths that start with s, ψ is satisfied until φ can be satisfied.
- $(M, s \vDash \psi\mathbf{AR}\varphi)$ iff $(\forall\pi$ such that $\pi_0 = s, \exists i \geqslant 0$ if $(\forall 0 \leq j < i(M, \pi^j \nvDash \psi))$, then $(M, \pi^i \vDash \varphi))$, for all paths that start with s, if ψ is not satisfied then φ will be satisfied.
- $(M, s \vDash \mathbf{EX}\psi)$ iff $(\exists\pi$ such that $\pi_0 = s, M, \pi^1 \vDash \psi)$, there exists a path that starts with s, next time ψ can be satisfied.
- $(M, s \vDash \mathbf{EF}\psi)$ iff $(\exists\pi$ such that $\pi_0 = s, \exists i \geqslant 0$ such that $M, \pi^i \vDash \psi)$, there exists a path that starts with s, eventually ψ can be satisfied.
- $(M, s \vDash \mathbf{EG}\psi)$ iff $(\exists\pi$ such that $\pi_0 = s, \forall i \geqslant 0 M, \pi^i \vDash \psi)$, there exists a path that starts with s, forever ψ can be satisfied.
- $(M, s \vDash \psi\mathbf{EU}\varphi)$ iff $(\exists\pi$ such that $\pi_0 = s, \exists i \geqslant 0$ such that $(\forall 0 \leq j < i(M, \pi^j \vDash \psi)) \wedge (M, \pi^i \vDash \varphi))$, there exists a path that starts with s, ψ is satisfied until φ can be satisfied.
- $(M, s \vDash \psi\mathbf{ER}\varphi)$ iff $(\exists\pi$ such that $\pi_0 = s, \exists i \geqslant 0$ if $(\forall 0 \leq j < i(M, \pi^j \nvDash \psi))$, then $(M, \pi^i \vDash \varphi))$, there exists a path that starts with s, if ψ is not satisfied then φ will be satisfied.

Model checking technique has many advantages over testing and other formal methods. Compared to testing technique, model checking uses temporal logic to describe system properties as a specification, specially, the continuous properties. The property can be verified exhaustively, but testing technique is hard to construct a complete set of test cases for verifying if the system satisfies the property. On the other hand, model checking offers counterexamples to trace the execution path of the system, when a property is not satisfied. The counterexample also can be used to detect a bug of the system as a test case. Compared to other formal method, like theorem proving, model checking is easier to verify if a system property is satisfied. Because model checking only needs a system description and automatically checks properties. Proving method needs to cost a lot of time to prove manually. In addition, model checking can be used in system design phase, since it does not need a complete specification of the system. The system can be abstracted based on objective properties.

Due to the fact that model checking is based on the exhaustive searching of the system's state space, the state space explosion problem is difficult to eliminate for verifying a complex system [22]. With the development of model checking, there are many new methods to improve the efficiency of verification. Symbolic model checking is possible to verify systems with more than 10^{120} states [23], which is using a symbolic representation to express state transition graphs. The symbolic representation is proposed by McMillan [24] based on ordered binary decision diagrams (OBDDs) [25]. It assigns boolean values to the set of state variables, and describes transition relations using boolean formulas. McMillan also have developed SMV specification language for the description of finite state concurrent systems [24]. There are some model checker support SMV language as input to verify systems, such as NuSMV and Cadence SMV.

There is an effective way to reduce the number of states is partial order reduction technique for asynchronous systems [26, 27]. The main idea of reducing the size of state space is to select a subset of the paths that can interleave independently executed transitions [22]. Bounded model checking (BMC) is another way to reduce state space, which is widely applied to verify sequential software and concurrent software based on SAT or SMT technique. BMC verifies systems by searching execution paths within a bound k, and SAT and SMT solver can handle propositional satisfiability problems with a large number of variables [28].

Statistical model checking (SMC) is an innovative approach proposed as an alternative to avoid the exhaustive exploration of the state-space of the model [19]. SMC techniques is a trade-off between testing an formal verification. The main idea of SMC is to perform simulations of a system for finitely many runs, and give a statistical result whether the

system satisfies properties with some degree of confidence. This method effectively reduces time and memory to be used. Of course, in contrast to classical model checking, SMC does not guarantee a correct result with 100% confidence. SMC is applied to interactive, distributed and embedded systems, such as CSMA/CD protocol, wireless network systems and distributed adaptive real-time systems. There are many SMC tools, such as PRISM, PRASMA-LAB, Yemen, MRMC, COSMOS, SMC-UPPAAL.

2.2 Timed Automata

Automata theory [29] is used to describe and analyze behaviors or computations of hardwares and softwares. Automata are a abstracted mathematical models of systems, which have a close relationship with Kripke structures. A finite automaton has a finite set of states, a set of initial states, a set of final states, a finite labels on transitions instead of the labeling function on states, and at most one transition between any two states. But, neither automata nor Kripke structures can accurately describe continuous time of systems. Properties related time are essential for real-time systems, such as response times, which affects the security and reliability.

To cope with this problem, timed automata was proposed by Alur, Courcoubetis and Dill [30], which is an extension of automata with a finite set of real-valued *clocks*. A timed automaton can precisely state system behaviors with clock constraints. The clock constraint on a state indicates a scope of time-lapse; the clock constraint on a transition indicates a time condition at which the transition occurs or reset the clock. The definition of clock constraints and timed automata are given below [31, 32]:

Definition (Clock Constraints) Let X be a set of clock variables, the set $\Phi(X)$ of clock constraints ψ is defined by

$$\psi := x < c | x < c | x \leq c | x \geqslant | \psi_1 \land \psi_2, \text{ where } x \subseteq X \text{ and } c \in \mathbb{C}.$$

Definition (Timed Automata) A timed automaton is a tuple $A = (L, l_0, X, \Sigma, I, T)$, where

- L is a finite set of locations,
- $l_0 \in L$ is a set of initial locations,
- X is a finite set of clocks,
- Σ is a finite set of labels,

- $I : L \to \Phi(X)$ is a mapping to assign clock constrains to locations,
- $T \subseteq L \times \Sigma \times 2^X \times \Phi(X) \times L$ is a set of transitions. A transition $< l, a, \psi, \lambda, l' >$ is from location l to location l' labeled with a. ψ is a clock constraint over X that specifies when the transition is enabled. $\lambda \subseteq X$ is a set of clocks that are reset when the transition is executed.

Definition (Semantics of TA) Let $A = (L, l_0, X, \Sigma, I, T)$ be a timed automaton. The semantics of A is defined by a transition system $\mathcal{T}(A) =< Q, Q_0, \Sigma, \to >$, where

- Q is a pair (l, v), $l \in L$ is a location and $v : X \to \mathbb{R}^+$ is a clock assignment,
- q_0 is a set of initial states, $\{(l, v) | l \in l_0 \wedge \forall x \in X[v(x) = 0]\}$.

In the transition system $\mathcal{T}(A)$, there are two types transitions are delay transitions and action transitions defined as follows:

- A delay transition is written as $(l, v) \overset{d}{\to} (l, v + d)$, where $d \in \mathbb{R}^+$, if $\forall d' : 0 \leqslant d' \leqslant d \Rightarrow u + d' \in I(l)$.
- A action transition is written as $(l, v) \overset{a}{\to} (l', v')$, where $a \in \Sigma$, if there exists $t =< l, a, \psi, \lambda, l' > \in T$, such that $v \in \psi$, $u' = [\lambda := 0]v$, and $v' \in I(l')$.

The timed automata formally defines real time systems with strict clock constraints. There are some tools based on the timed automata theory, such as UPPAAL, RED, PAT, MRMC and so on. UPPAAL model checker has a user-friendly interface to model, simulate and verify a design model described in a network timed automaton. Moreover it also supports statistical model checking for verifying a probabilistic timed model. In order to improve the efficiency and capability of classical model checking, we choose UPPAAL model checker as our tool to check IVN system design model.

2.3 UPPAAL

UPPAAL [33, 34] is an explicit model checking tool with a nice interface developed by Uppsala University and Aalborg University. It is based on timed automata theory for modeling, simulating and verifying real-time systems. UPPAAL serves modeling language to describe system behaviors as a network of timed automata, and specification language to express system properties. It is efficient and practical to verify system properties by exhaustively exploring state space of the system. It has been applied successfully

in case studies ranging from communication protocols to multimedia applications. Since UPPAAL first came out in 1995 [35], it has been extented for special problems, such as UPPAAL-TIGA, UPPAAL-CORA, UPPAAL-SMV and UPPAAL-TRON. In this research, we employ original UPPAAL to do classical model checking, and UPPAAL-SMC to do statistical model checking.

2.3.1 Modeling Language

UPPAAL provides both graphical and textual formats for modeling systems. The graphical format is a state transition graph based on timed automata. The textual format provides a basic programming language for timed automata [34]. To enrich expressions of the state transition graph, there are four kinds of labels for a transition in UPPAAL [33].

- **Select**: a select label is expressed in `<variableName>:<type>`, which assigns a non-deterministic value under the `type` for the variable. For example, `msg: msgID` says that a message identifier is selected from a specific range of `msgID`.
- **Guard**: a guard label is a boolean expression about clocks, integer variables and constants. If and only if the expression is true, transition can be executed. For instance, a label `x > 5` is a clock constraint between two locations l and l'.
- **Synchronization**: a synchronization label is in the form of `exp!` or `exp?`. They have to appear in pair in a network of timed automata, which are evaluate to a channel and serve for synchronization between timed automata. Only when `exp!` was happened in an automaton, the corresponding transition with `exp?` will be performed in another automaton.
- **Update**: a update label allows reset clocks, integer variables, and constants and executes function to assign them.

These labels accurately describe transition relations between locations. UPPAAL also provides three kinds of locations that present time-delays [33].

- **Normal location**: a normal location allows time lapse and it may or may not have an invariant label that is a clock constraint. If there is no expression for a clock in the normal location, the value of the clock is unknowable. If there is a clock constraint in the normal location, the value of the clock has to satisfy the constraint in the location.

- **Urgent location**: there is no time-lapse in the urgent location, that is, all clocks are frozen. Similarly, the location adds a clock constraint `x<=0`.

- **Committed location**: A committed location does not have time lapse, and has to have a transition to leave this location.

2.3.2 Specification Language

Model checkers are used to verify if a formalized model of a system meets the specification of the system. The specification language in UPPAAL is called query language, which is a subset of timed computation tree logic (TCTL) [36]. The query language can describe both states and paths of the model. Properties of the system are written in the query language to verify satisfiability of the system model. There are five kinds of temporal properties provided as follows, where `p` and `q` are properties.

- **Possibly**: `E < > p`

  The property is true iff there is a transition path $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n$ in a timed transition system, where s_0 is the initial state and s_n satisfies `p`. That is, there is a path that satisfies the property `p` in a certain state.

- **Invariantly**: `A [ ] p`

  The property is true iff all reachable states satisfy `p` in a transition system, which is equivalent to `E < > not p`.

- **Potentially always**: `E [ ] p`

  The property is true iff there is a transition path $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_i \rightarrow ...$ where `p` is satisfied for all state s_i.

- **Eventually**: `A < > p`

  The property is true iff all possible paths eventually reach a state satisfying `p`, which is equivalent to `not E[ ] not p`.

- **Leads to**: `p → q`

  The property means whenever `p` holds eventually `q` will hold as well, which is equivalent to `A[](p implies A<>q)`.

Deadlocks is also a important property for real-time systems. In a deadlock state, there is no outgoing transitions neither from itself or its delay successors.

2.4 UPPAAL-SMC

2.4.1 Modeling Language

The UPPAAL-SMC is an extension of UPPAAL based on a stochastic interpretation and priced timed automata (PTA) [37, 38]. The stochastic interpretation as an alternative describes non-deterministic behaviors of systems using probabilistic choices. The time-delays for the non-deterministic behavior to occur are defined by probability distributions. There are two kinds of probability distributions are provided in UPPAAL-SMC, including uniform distributions and exponential distributions.

- **Probabilistic choices**: it means that a location has multiple transitions to next location, which transitions are non-deterministic according to probabilities. The probabilities are stated by a label **Probability weight** on branch transitions.
- **Uniform distributions**: it means that the time-delay accords with a uniform distribution in a location with a time-bounded delay. The time-bounded is described by clock constraints in the label **Guard**.
- **Exponential distributions**: it is a rate parameter of exponential density function, which is set on a location without time-bounded delays by a label **Rate of exponential**. The rate indicates that the time-delay in a location conforms to a exponential distribution with the rate designed by users.

2.4.2 Specification Language

The specification language in UPPAAL-SMC is based on weighted metric temporal logic (WMTL) [39]. UPPAAL-SMC provides five queries for verifying probabilistic properties [40], where `p` and `q` are expression of properties, `pro` is a probability, `N` indicates the number of simulations to be performed, and `B` is a time bound for the simulation.

- **Statistical evaluation**: `Pr[<=N](< >|[ ] p)` This query estimates the probability of the state property.
- **Hypothesis testing**: `Pr[<=N](< >|[ ] p) <=|>= pro`

  This query states whether the probability of the state property is satisfied within a certain probability `pro`.
- **Statistical comparison**: `Pr[<=N](< >|[ ] p) <=|>= Pr[<=N](< >|[ ] q)`

  This query is to compare the probabilities of property `p` and `q`.

- **Expected value**: `E[<= N;B] (min|max:  p)`

  This query is to evaluate expected value of the maximal or the minimal value of a property `p`.

- **Simulations**: `simulate M[<=N]{p,q}`

  This query is used to simulate a system in `M` times and compute trajectories of specified properties over time.

2.5 In-vehicle Networking System

IVN systems is a kind of special internal communication networks by connecting inside components of an vehicle. The communication network requires software and hardware that comply with certain standards. Software is standardized by founders, such as AUTOSAR and OSEK/VDX. AUTOSAR and OSEK/VDX were introduced to a variety of products, which founded by German automotive industry, OMS, AUDI, BMW, Volkswagen, etc. On the hardware side, the major standards is LIN, CAN, FlexRay, MOST, and Ethernet has also started to enter automotive fields. The transmission medium of LIN, CAN, FlexRay, and Ethernet is copper wires and twisted pair wires is used except LIN. MOST uses optical fibers as transmission medium, that are not susceptible to electromagnetic impact, but expensive and fragile. CAN has the longest history in the global automotive industry, especially in power systems. LIN is a low cost and speed solution to control simple devices, such as doors, windows, seats and so on. FlexRay is designed for safety-critical systems [41], faster and more reliable than CAN. MOST is mainly used in entertainment system. Ethernet is used for vehicle diagnostics in the garage.

The IVN system has strict requirements on real-time, safety and reliability. Also, vehicle producers have to consider cost, applicability and so on. As a result, a mixed network system can flexibly meet the requirements of various control system in an automobile. In this work, we mainly regard CAN and FlexRay protocol. The CAN protocol is the most widely used, and suitable for soft real-time system, such as engine management, anti-lock brakes, and cruise control. The FlexRay protocol is a deterministic and fault-tolerant protocol for safety-critical systems. Moreover, these two protocols have different communication mechanisms. Above all, some basic information of CAN and FlexRay are listed in Table 2.1. Then we will introduce the two protocols and their communication modes by two examples respectively.

Table 2.1: CAN and FlexRay comparison

No.	Item	CAN	FlexRay
1	Max. transfer rate	1 Mbit/s	10 Mbit/s
2	No. of channel	1 channel	2 or 1 channel
3	Network topology	Bus	Mix. of bus and star
4	Architecture	Multi-master up to 40 nodes	Multi-master up to 64 nodes
5	Communication mechanism	CSMA/CA	TDMA/FTDMA
6	Message identification	Identifier	Time slot
7	Data fileds	8 bytes	254 bytes
8	Application	Soft real-time	Hard real-time
9	Example	Engine, Anti-lock break	Powertrain, Chassis

2.5.1 Controller Area Network

The CAN protocol [42] is an event-triggered control network, developed by Bosch in the early 1980s. It provides efficient and secure support for data communication in distributed real-time control systems [43]. Its application domain ranges from high-speed networks to low-cost multiplex wiring. There is a series of CAN bus communication protocol, such as CAN-A, CAN-B, TTCAN and CAN-FD. They are suitable for different systems and applications. Such protocols only define the two lowest layers, data link layer and physical layer of the OSI model. In order to use CAN protocol, the higher layer, application layer, also need to be standardized, such as SAE J1939, CANopen and DeviceNet. Besides, the CAN protocol offers error detections and error handlings to guarantee the reliability and safety of communications. Therefore, it is not only widely used in automotive industry, but also automatic control, mechanical industry, aircraft industry, etc..

In the automotive industry, CAN is used for mainstream powertrain communication systems with bitrates up to 1 Mbit/s and low-cost body control systems [44]. CAN systems usually use bus topology to connect each node through single channel. The number of nodes is up to 22 in a CAN system. The CAN protocol adopts multi-master broadcasting method to transfer messages and synchronizes time between each node. That is, CAN nodes broadcast their messages to all connected nodes concurrently, and each receiving node may independently processes the messages. The CAN uses a carrier sense multiple access/collision detection (CSMA/CD) access control method to avoid multiple nodes accessing bus in the same time. CAN frames have an identifier denoting its priority

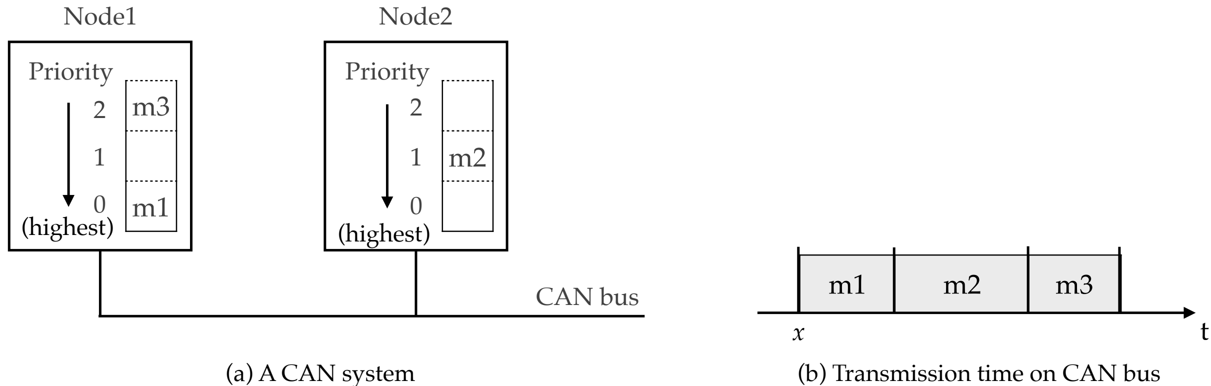


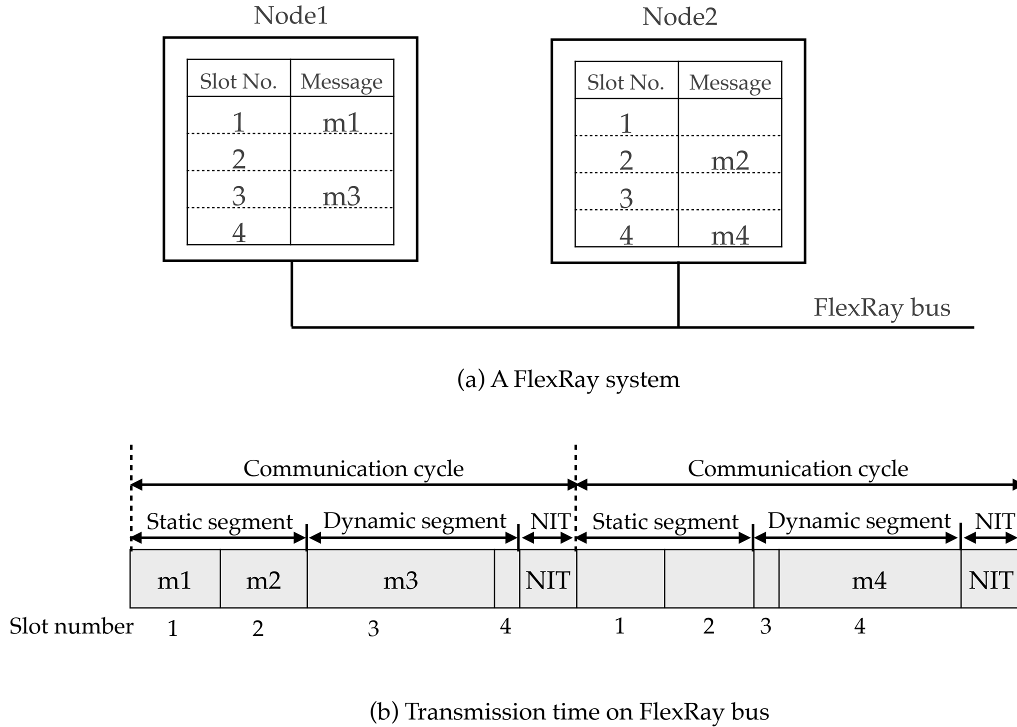
Figure 2.2: Example of CAN message transmission.

and a data field of 0–8 bytes, and all the frames participate arbitration based on a fixed priority scheduling algorithm. The frame with the highest priority is sent once the bus is free. Other frames wait for the next arbitration period.

Fig. 2.2 shows an example of the message transmission scheme in CAN. A CAN system has two nodes, Node1 and Node2, as shown in Fig. 2.2 (a). At time `x`, CAN bus is free, and the CAN system has three message ready for transmission. There are two messages, `m1` and `m3`, in transceiver buffers of the Node1, and a message `m2` in a transceiver buffer of the Node2. Assume that there is no new messages to participate in the arbitration during the message transmission. The arbitration process determines that `m1` has the highest priority, and so this message is immediately sent to the bus first. Once `m1` has been sent, no other message is stored in the buffer. In the next arbitration period, `m2` has the highest priority and is sent to the bus. At last, `m3` is arbitrated successfully and sent to the bus.

2.5.2 FlexRay

FlexRay has been especially developed by the FlexRay consortium since 2000 for safety related applications in the vehicle industry [45]. It is applied in real-time applications and as a replacement of CAN when higher data rates are required. FlexRay supports X-by-Wire applications and has been used in safety-critical system, such as steer-by-wire and brake-by-wire [46, 47, 48]. FlexRay is a high-speed, flexible communication protocol, and offers excellent fault-tolerance computing. It has two communication channels with a data rate of 10 Mbit/s. FlexRay data frames contain fields of 0–254 bytes. Furthermore, the communication scheme of FlexRay is time triggered to ensure a defined communication time and clock synchronization for all nodes. A FlexRay system consists of several master nodes and two communication channels for providing reliable communication. To reduce cost using only one channel can be sufficient. FlexRay networks use a star, a bus or a



(a) A FlexRay system

(b) Transmission time on FlexRay bus

Figure 2.3: Example of FlexRay message transmission.

mixed topology for constructing IVN systems.

The communication of FlexRay not only supports static time division multiple access (TDMA) scheme, but also a dynamic mini-slotting-based scheme based on communication cycles. In TDMA networks, each message is transmitted in a certain time slot. That is, messages are able to access bus during their own time slots. So message transmission is predictable and determinable. However, the bus bandwidth cannot be used effectively. The dynamic mini-slotting-based scheme allocates time slots dynamically. If any transmission happens within a mini-slot, the time slot will be expanded until it meets the required mini-slot of the message. Thus, the utilization ratio of the bus is increased. In FlexRay networking systems, message transmission takes place in periodic communication cycles, each of which involves a static segment and a dynamic segment. The static segment employs TDMA scheme, which is divided into static slots. A static message is assigned to a fixed static slot. The dynamic segment employs the dynamic mini-slotting-based scheme, which is divided into mini-slots. Dynamic messages are transmitted according to their priority, and take several mini-slots. Those unused mini-slots will serve as network idle time (NIT) in the communication cycle.

Fig. 2.3 demonstrates a FlexRay system, which has two nodes, and there are two communication cycles for representing FlexRay message transmission. Each cycle has

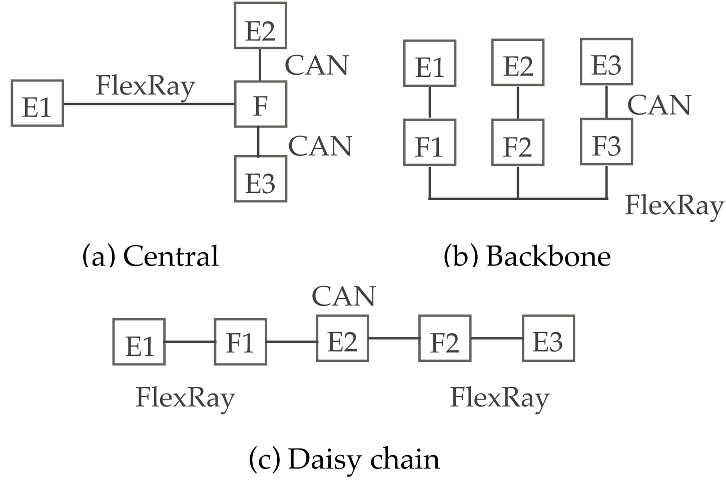


Figure 2.4: Three topologies of IVN systems.

four slots, two static slots `slot1` and `slot2` and two dynamic slot `slot3` and texttttslot4. The message identifier is set to the slot number for convenience. In the Node1, message `m1` is assigned to static slot `slot1` and message `m3` is assigned to dynamic slot `slot3`. In the Node2, message `m2` is assigned to static slot `slot2` and message `m4` is assigned to dynamic slot `slot4`. Assume that there is no other message transmitted during the two communication cycle. `m1` and `m2` should be sent in `slot1` and `slot2` within the first communication cycle. Although `m1` and `m2` is not sent in the second cycle, the time interval of `slot1` and `slot2` still elapse with no transmission. `m3` and `m4` should be sent in `slot3` and `slot4`, and the length of these dynamic slots relys on the length of the message. After `m3` has transmitted in the first communication cycle, there is no enough mini-slots for transmitting `m4`. So the `m4` was not sent in the first cycle, and `slot4` occupies one mini-slot. If the maximum number of slots is reached, there is still some mini-slots, the NIT will start, which is no transmissions until the end of that communication cycle. The remaining `m4` will wait for the second communication cycle and it will be sent in the `slot4`.

2.5.3 Topologies of IVN systems

IVN systems have varied and complicated topologies, in which subsystems with different protocols are connected by gateways. We considered three typical topologies based on gateways, central, backbone and daisy chain [8]. They are introduced in the following (see Fig. 2.4):

- The central topology only has a single gateway, and nodes/subnetworks connect through the gateway. All messages that need to be transmitted need to be routed through the gateway. If too many messages are blocked in the gateway, the gateway may become congested, a lot of network delay or even data loss will be caused.

- The backbone topology is that multiple gateways connected by a bus, following a single protocol, and each gateway connects with some nodes following other protocols, as a subnetwork. All transmissions between different subnetworks have to be forwarded twice by two gateways. The advantage of this topology is that the transmission time is not affected by the number of nodes, but it is related to the amount of messages on the gateway bus.

- The daisy chain topology consists of several gateways and nodes/subnetworks in an alternating chain. The response time is a function of the distance between environments. The number of messages also influences the response time, as many messages may cause network congestion.

Chapter 3

Abstractions of IVN Systems

An IVN system is not only composed of multiple nodes, but each node also needs to conform to corresponding communication protocol specification and operating system standards. Furthermore, these specifications and standards have a number of parts that are irrelevant in terms of verifying the communication behavior of IVN design models. The verification of properties is particularly difficult with state space problem. Hence, it is necessary to make an appropriate abstraction for IVN systems. We give a two-staged abstraction, abstracting the architecture of the system and the composition of protocol specifications.

3.1 Abstracting the Architecture of IVN Systems

A practical IVN system is composed of many nodes, buses and gateways as shown in the upper part of Fig. 3.2. There are two kinds of nodes and buses, and the blue square and blue line represent a CAN node and a CAN bus, and the green square and green line represent a FlexRay node and a FlexRay bus. The red square represents a gateway. We define every component of the system for abstracting the architecture of IVN systems as follows:

- Nodes: a node is an ECU, which is used for data processing, and sending-receiving data.
  - CAN nodes: a CAN node is a node that sends and receives data according to the CAN protocol.
  - FlexRay nodes: a FlexRay node is a node that sends and receives data according to the FlexRay protocol.

- Buses: a bus is used to connect nodes and provides communication standards for transmitting data.
  - CAN buses: a CAN bus is used to connect CAN nodes and provides CAN protocol for transmitting CAN messages.
  - FlexRay buses: a FlexRay bus is used to connect FlexRay nodes and provides FlexRay protocol for transmitting FlexRay messages.
- Subnetworks: a subnetwork is composed of one or more nodes connected together by a kind of bus, which conform to the same communication standard as the bus.
  - CAN subnetworks: a CAN subnetwork is composed of one or more CAN nodes connected by CAN buses.
  - FlexRay subnetworks: a FlexRay subnetwork is composed of one or more FlexRay nodes connected by FlexRay buses.
- Gateways: a gateway is used to connect multiple subnetworks and supports CAN buses and FlexRay buses for message transmission. Gateways can be connected together in a bus.

An IVN system has several subnetworks with different protocols, and they communicate with others through gateways. The subnetwork is composed of multiple nodes connected in a topology, such as point to point topology, star topology and hybrid topology, as shown in Fig. 3.1. These subnetwork are combined by at least one gateway in an IVN system.

Within a subnetwork, there is also message transmission between nodes and such messages can be verified with a single protocol, but this work focuses on the communication between the subnetworks using heterogenous protocols. Therefore, the first stage abstraction is to simplify the architecture of IVN systems. The subnetwork is replaced by an environment with its original communication protocol. We ignore the topology of the subnetwork and internal communications, and focus on its communications with external subnetworks. Environments have two types, CAN environments and FlexRay environments. Every environment holds one or more tasks and is connected to gateways by CAN bus or FlexRay bus. In addition, we keep the same gateway topology of the IVN system, as we describe in subsection 2.4.3.

For abstracting the architecture of an IVN system, first, we need to find all gateways in the IVN system, and then separate the system from the gateways. There are subnetworks

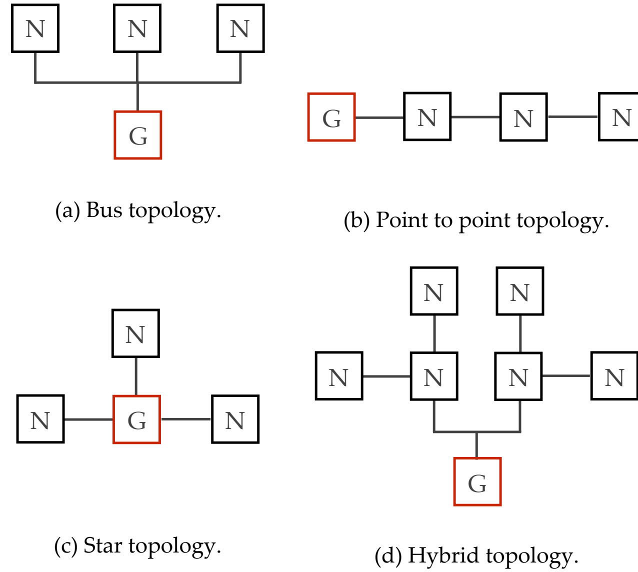


Figure 3.1: Subnetwork topologies

connected the gateways through a bus, or between gateways. These subnetworks are abstracted as corresponding environments, and then these environments are connected to gateways which the original subnetworks was connected by corresponding buses. Since the gateway is not the focus of our research, we consider forwarders instead of gateways for forwarding messages between environments. The buses are communication standards according to protocol specifications, therefore, we define bus protocols in the abstracted IVN system. We ignore the physical connections between nodes and this work is centered on communication rules based on protocol specifications. By the abstraction, the architechture of the IVN system is shown in the bottom of the Fig. 3.2 (b). The blue squares and green squares with E are CAN environments and FlexRay environments. They are connected to the corresponding forwarders as the above gateways. The blue and green dotted lines represent CAN protocol and FlexRay protocol. An abstracted IVN system will consist of the following components.

- Environments: an environment is an abstracted subnetwork, which is responsible for sending messages from the original subnetwork to other subnetworks and receiving messages from other subnetworks to it.
  - CAN environments: a CAN environment is an abstracted CAN subnetwork that has tasks run on an ECU to send or receive CAN messages to other environments.

- FlexRay environments: a FlexRay environment is an abstracted FlexRay subnetwork that has tasks run on ECU to send or receive FlexRay message to other environments.

- Forwarder: a forwarder connects multiple environments by bus protocols according to the original topology. It supports forwarding messages between CAN environments and FlexRay environments. The forwarder is a task that implements interpreting and scheduling frames between different protocols.

- Bus protocols: a bus protocol carries out transmitting behaviors based on protocol specifications.

  - CAN bus protocol: CAN bus protocol is used to connect CAN environments and forwarders, and responsible for transmitting CAN messages. The CAN bus protocol is corresponding with the layer structure of ISO/OSI model defined by CAN specification, including the object layer, transfer layer, and physical layer. These layers are described in the previous section 3.1.1.
  - FlexRay bus protocol: FlexRay bus protocol is used to connect FlexRay environments and forwarders, and responsible for transmitting FlexRay messages. The FlexRay bus protocol is corresponding with the FlexRay protocol specification. Although FlexRay specification does not explicitly provide a layer structure model, this model is introduced in some related studies [**?**, 49]. We will introduce the FlexRay bus protocol by referring to the layer structure model in section 3.1.2.

Although we have simplified the system structure and reduced the number of nodes, the communication standards in the system are still complex. So we did the next stage of abstracting communication protocols.

3.2 Abstracting the Composition of Protocol Specifications

The abstract result of an IVN system has CAN environments and FlexRay environments. Although these environments are abstraction of subsystems, their communication behaviors still need to conform to protocol specifications. The CAN specification and FlexRay specification respectively describe the communication behaviors in details [42, 45]. However, a lot of details will result in an enormous number of states for the system, and make

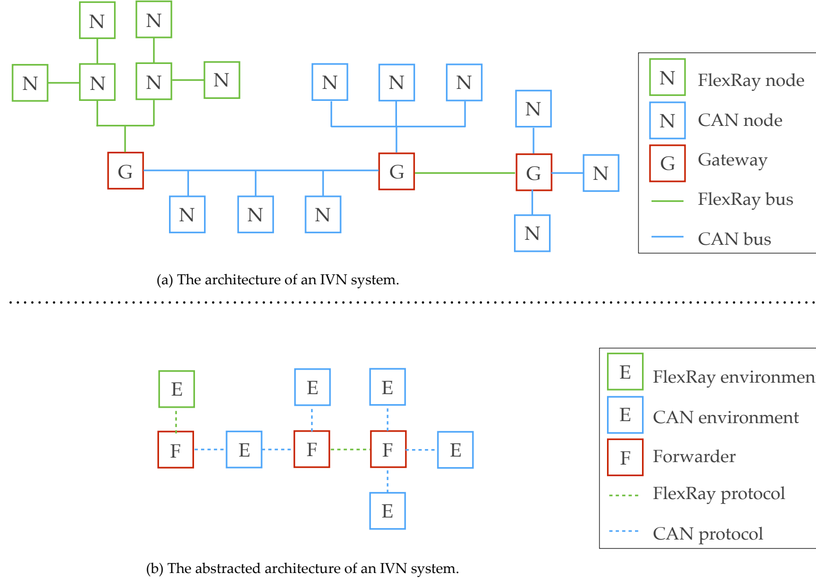


Figure 3.2: Abstraction for the architecture of IVN systems.

it difficult to check. Whereas if we simplify the system too much, it is hard to verify properties of the system availably. Therefore, an proper abstraction of the specifications is significant for checking IVN systems. We will briefly introduce the overview of CAN and FlexRay specification, and give an abstracted composition of protocol specifications in this section.

3.2.1 Overview of Protocol Specifications

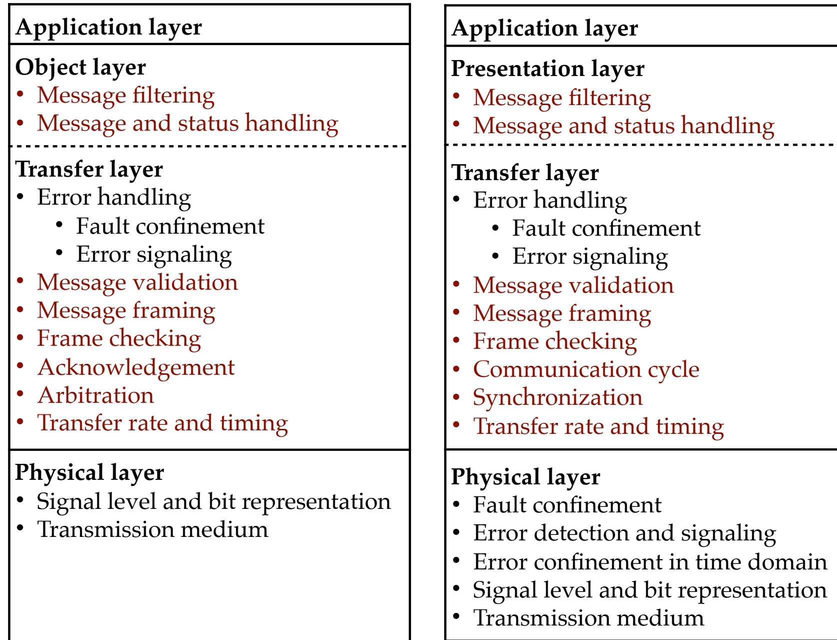
The CAN and FlexRay specifications state the layer structure of nodes based on ISO/OSI model, and elaborate on each layer.

CAN Specification

The CAN specification defines physical layer and data link layer of ISO/OSI model, where the data link layer consists of objective layer and transfer layer [42]. Fig. 3.3 (a) shows the layered structure of the CAN protocol and functions in each layer.

- The *Application layer* was not defined in the CAN specification, but is standardized by founders, such as AUTOSAR and OSEK/VDX. This layer performs specific tasks to realize control functions.

- The *Object layer* provides logical link control of the data link layer. The principal function of the *Object layer* is to filter messages and handle the message status.



(a) CAN protocol. (b) FlexRay protocol.

Figure 3.3: Layered structures of the CAN protocol and FlexRay protocol.

- The *Transfer layer* is the core of the CAN protocol, performing media access control (MAC) of the data link layer. It controls the process of messages access to bus and ensures the correctness of message transmissions. There is an error handling mechanism to detect errors in the bit levels and handle them appropriately, such as shutting down defective nodes and retransmitting messages. In addition, this layer is responsible for clock synchronization. It checks and distinguishes messages, then determines which message is to be sent through the arbitration process.

- The *Physical layer* defines behaviors related to how signals are transmitted in the physical medium.

FlexRay Specification

FlexRay specification gives explanations of communication mechanism in specification and description language (SDL) [45]. The layer structure of the FlexRay protocol is defined as shown in Fig. 3.4. The FlexRay consists of a *Controller Host Interface (CHI)*, a *Communication Controller (CC)* and a *Bus Driver*. The CC is divided to six parts as follows:

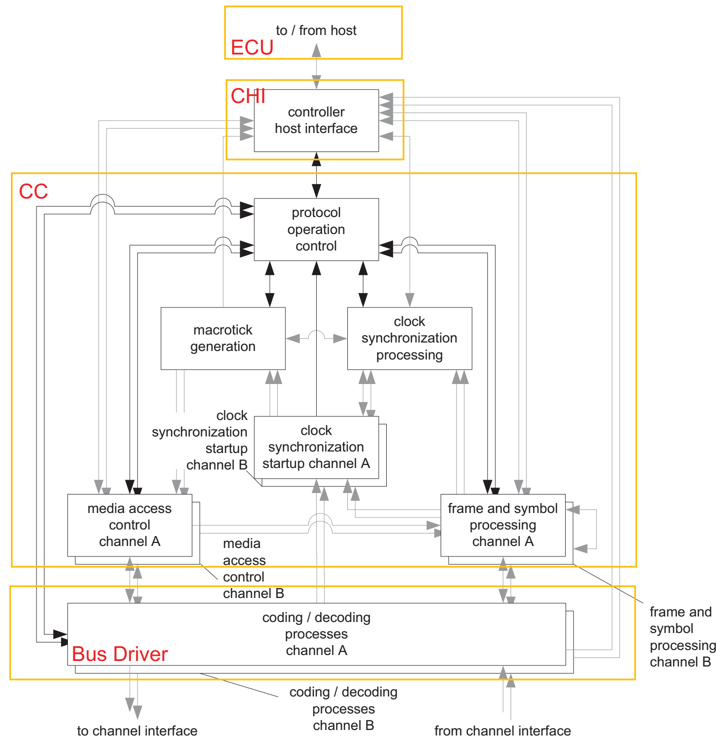


Figure 3.4: Architecture of the FlexRay protocol.

- Protocol operation control (POC) controls all states of the node, and shares its states to the host through CHI. The POC performs the host commands and manages the CC.

- Media access control (MAC) controls communication cycle and transmit frames corresponding to two communication mechanisms, TDMA and dynamic mini-slot based TDMA.

- Frame and symbol processing (FSP) takes charge of checking the correctness of frames and symbols, when frames and symbols are received during a special slots.

- Macrotick generation (MTG) provides counters of the number of communication cycles and macroticks, and applies offset values and correction values.

- Clock synchronization processing (CSP) synchronizes the clocks of every node by sending a synchronization frame. The CSP detects deviation values of clocks between each node, and compute the offset of each slot and the rate correction values.

- Clock synchronization startup processing performs the initialization and starts MTG

and CSP processes.

Although FlexRay specification does not explicitly present a layer structure of ISO/OSI model, the structure is introduced in some related studies [49]. The layered structure of the FlexRay protocol is similar to the CAN protocol, as shown in Fig. 3.3 (b). The *Application layer* is also not given specific functions, which contains programs developed by the application developers. The *Presentation layer* is in charge of message filtering and status handling. The *Transfer layer* defines the communication cycle and transfer rate and timing, and realizes clock synchronization. It validates messages and frames and has an error handling mechanism. The *Physical layer* not only defines signal transmission in the physical circuit, but also detects errors in the time domain.

According to the architecture in the FlexRay specification and the layer structure in the related studies, we give out their correspondence as follows:

- The *Application layer* has tasks that run on a real-time kernel, as the ECU in the FlexRay specification.
- The *Object layer* functions are similar to the CHI in the FlexRay specification. It stores messages and controls interaction between the ECU and CC instead of the CHI.
- The *Transfer layer* carries out communication behaviors and is responsible for handling errors and clock synchronization, corresponding to the CC.
- The *Physical layer* takes charge of representing frames on signal level, and handling errors that occur on hardware, as the Bus Driver in FlexRay specification.

3.2.2 The Composition of protocol specifications

The CAN specification and FlexRay specification not only provide the layer structure, but also describe the specific functions in each layer in detail, such as frame format, transmission process, error handling, clock synchronization and so on, which contains a lot of information about how to implement the protocols on hardware. However, the purpose of our research is to verify the IVN system in design phase. For verifying the system at design phase, behaviors associated with the physical layer is not necessary, so the underlying implementation of the system is not involved in the system design model. Therefore, we abstract the protocol specifications to remove hardware related contents in each layer, and ignore the *Physical layer*. On the other hand, since this research aims to verify reachability and timed property of message transmission between different

Application layer
- User-developed applications

Interface layer
- Message filtering
- Message and status handling

Communication control layer
- Message validation
- Message framing
- Frame checking
- Transmission schemes
  - CAN transmission scheme
  - FlexRay transmission scheme
- Transfer rate and timing

Medium layer
- Transmission medium
  - CAN Medium
  - FlexRay Medium

Figure 3.5: The common structure of the CAN specification and FlexRay specification.

protocols without corrupted frames and error signals, error detection and error handling of the specifications do not work in the node. So we do not consider the error handling function in the *Transfer layer*. In Fig. 3.3, we have highlighted the functions we care about in red font, leaving out the functions in black font.

Moreover, comparing the layer structure of the CAN and FlexRay protocol, except for a few differences in the *Transfer layer*, the *Object layer* of the CAN and the *Presentation layer* of the FlexRay have same functions. We can know that they have the same layer structure, and each layer has similar functions. Therefore, we propose a common structure for CAN protocol and FlexRay protocol that consists of the *Interface layer*, *Communication control layer* and *Physical layer* as shown in Fig. 3.5.

- *Application layer*: the specifications of the CAN and FlexRay do not define concrete functions of the *Application layer*, because the *Application layer* executes diverse tasks developed by users on an ECU, as the environments we abstracted.

- *Interface layer*: the specifications of CAN and FlexRay have the same functions in the *Object layer* and *Presentation layer*, which is filtering messages and handling messages and status. Furthermore, they connect the *Application layer* and *Transfer layer*, and are important interface for exchanging data between nodes. Hence, we use an *Interface layer* to keep the original functionality.

- *Communication control layer*: CAN protocol and FlexRay protocol adopt completely different ways to transfer messages, CAN protocol is based on event-driven and FlexRay protocol is based on time-driven. The *Tranfer layer* of CAN and FlexRay perform transmission behaviors to control communication between nodes. To uniformly describe the composition of CAN and FlexRay environments, we define a *Communication control layer* instead of the *Tranfer layer* of CAN and FlexRay. This layer retains the same functionalities and abstracts a transmission scheme to express the unique functions of CAN and FlexRay environments, so the layer keeps communication schemes defined by the specifications, message validation, frame checking and transfer rate and timing. If there is a CAN bus, the *Communication control layer* will adopt the CAN transmission scheme to communicate. If it is a FlexRay bus, the *Communication control layer* will adopt the FlexRay transmission scheme to communicate.

- *Medium layer*: though we ignore all functions in *Physical layer*, it still needs a transmission bus to represent the transmission of messages over hardware. So we abstract the bus to store the frame transmitted on hardware. If there is a CAN bus, the *Medium layer* will be a CAN medium. If it is a FlexRay bus, the *Medium layer* will be a FlexRay medium.

In the common structure of CAN and FlexRay protocols, the *Communication control layer* is the core of IVN systems and the most important part of our IVN system design model. A large chunk of the specifications of CAN and FlexRay is devoted to their *Transfer layer*, and these descriptions are mixed with a lot of hardware implementation and error handling that the abstraction of the specifications becomes much more difficult. We carefully read the protocol specifications and manually build communication control models for CAN and FlexRay. We will introduce how to pick up needful information to build abstracted models for the *Communication control layer* in next section.

3.3 Communication Control Models

Because of the differences of transmission schemes, we can only establish two independent models for each CAN protocol and FlexRay protocol. However, the description of the transfer layers is mixed with a lot of signal processing and implementation on the hardware in the CAN specification and FlexRay specification. We abstracted the communication protocols based on fault-free communication, mentioned the second abstraction stage. The communication control models should meet our verification requirements and be as

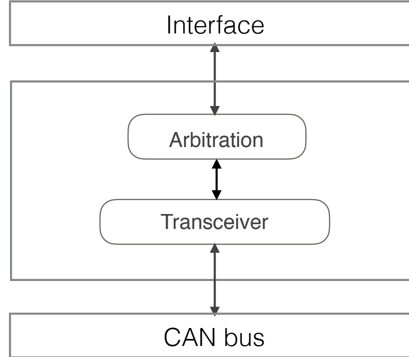


Figure 3.6: Structure of the CAN communication controller.

brief as possible. The *CAN* and *FlexRay* models in UPPAAL will be explained as the following sections.

3.3.1 CAN Communication Control Model

The CAN system adopts multi-master broadcast access to transmit messages. Any node may send a message when the bus is free. The node, which has the message with the highest priority, obtains the right to access the bus. To ensure that the message being transmitted has the highest priority, there is bitwise arbitration using the identifier of messages. The identifier has 11 bits, which indicates the priority from 0–10 (ID-0 is the highest priority.). During the arbitration, all nodes requesting to send a message have to compare their identifiers bit by bit. The message with highest priority gains the bus. In our model, we use an integer instead of message identifier because the expression of bit level is ignored.

The *CAN* model mainly realizes the CSMA/CD mechanism, that is a message arbitration process. We implement two automata to represent CAN protocol model, an *Arbitration* automaton and a *Transceiver* automaton. The relation of the two automata is shown in Fig. 3.6. The *Arbitration* monitors CAN *Interface* in real time, and determines which message can be transmitted. The *Transceiver* is responsible for delivering the corresponding message to the CAN transmission medium, and stores the message to the CAN *Interface* when the transmission ends.

The *Arbitration* automaton is shown in Fig. 3.7. It monitors whether a CAN task send a transmission request in real-time. If the *Arbitration* supervises a request, a synchronization channel `transmissionRequest`, it will scan buffers from the highest priority identifier (`index=1`) to the lowest message priority (`index=maxPriority`). A message with higher priority wins the arbitration process. The arbitration result is synchronized

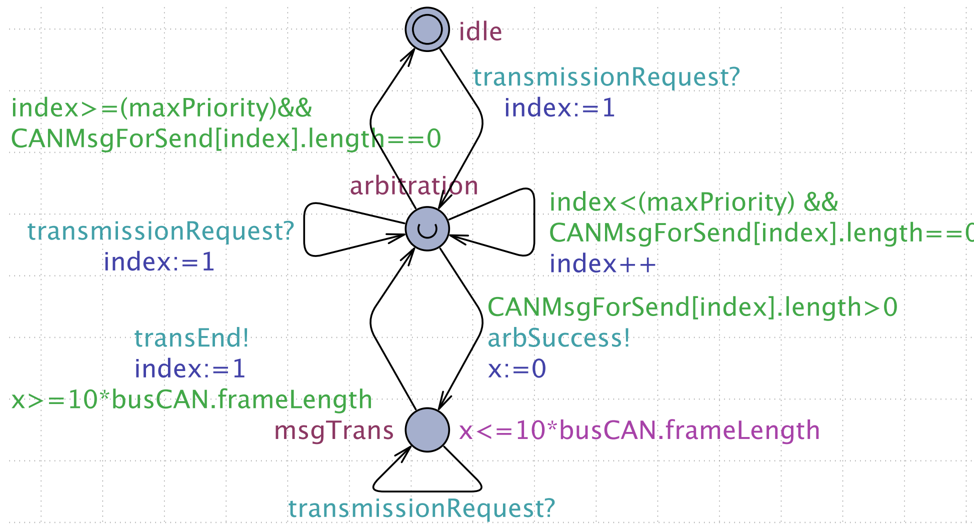


Figure 3.7: Arbitration automaton.

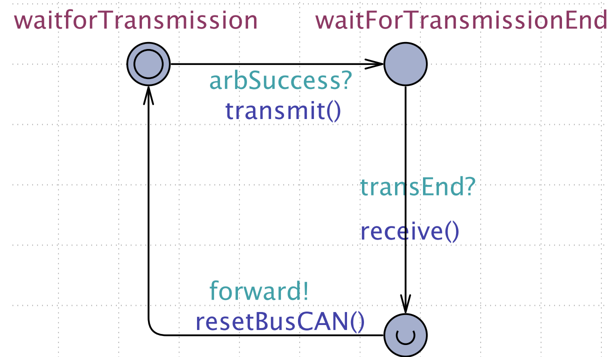


Figure 3.8: Transceiver automaton.

to the *Transceiver* by the channel `arbSuccess`. The *Transceiver* automaton leaves the location `waitforTransmission`, as shown in Fig. 3.8. The *Arbitration* waits in the location `msgTrans` for the message transmission time given by a value *msgLength*. When the message transmission is completed, the *Transceiver* synchronizes with the *Arbitration* using the channel `transEnd`. The *Transceiver* stores the message to the CAN *Interface* and resets the buffer `CANMsgForSend[index]` by the function `receive()`, and clears the CAN bus by `resetBusCAN()`. Then the *Arbitration* will restart arbitration from the highest priority identifier. The channel `received` is used to synchronize with a gateway model or a receiver model.

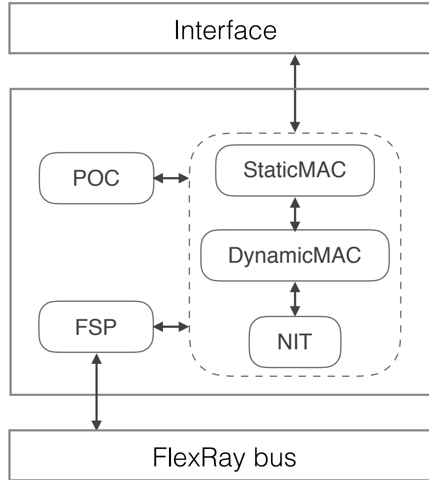


Figure 3.9: Structure of the FlexRay communication controller.

3.3.2 FlexRay Communication Control Model

The FlexRay specification defines two types of communication schemes within a communication cycle, the static TDMA and dynamic minislot-based TDMA. The communication cycle contains of static segment and dynamic segment to transmit static messages and dynamic messages. As shown in Fig. 3.4, the *FlexRay* communication controller contains six parts, where the CSP and clock synchronization startup processing are responsible for synchronization between nodes, the MTG is in charge of counting the number of cycles, slots and macroticks, and provides the offset and rate. We abstract the protocol specification under assuming fault-free, so all nodes are synchronous and correct. Consequently, we use a global clock to synchronism all nodes in the FlexRay system. Also, we create some variables instead of the MTG. The FlexRay MAC is much more complicated than CAN, since it adopts two communication schemes, static MAC and dynamic MAC, and perform the communication cycle. As a result, the FlexRay communication controller is abstracted into five automata, as shown in Fig. 3.9. Each automaton will be introduced according to FlexRay specification.

Protocol Operation Control

The *POC* controls protocol operation following host commands from ECUs, and manages status of nodes, which has 6 basic states. When applications in the ECU are executed, the states of nodes have to be changed by the *POC*. The protocol operation control process is shown in Fig. 3.10.

- In the *Default config* state, *POC* waits for distinct commands from *ECU*, and en-

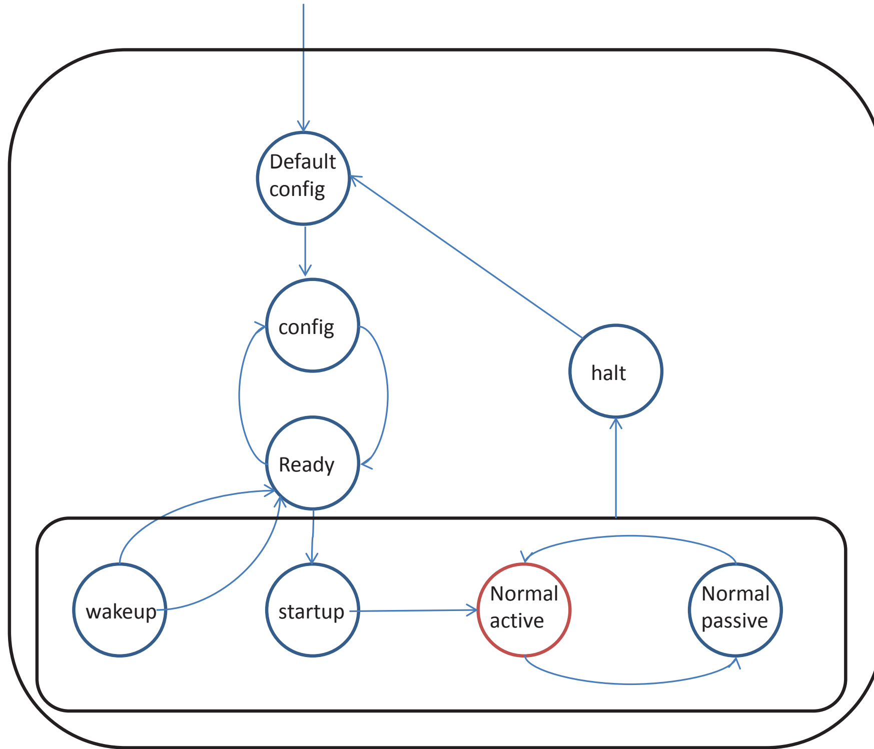


Figure 3.10: Overview of protocol operation control.

sures initialization data of the communication controller, such as the communication cycle and data rate.

- In the *Config* state, *POC* executes initialization of the communication controller. When the initialization is completed, the *POC* moves to *Read* state.

- The *Ready* state indicates that the communication mechanism has been created for starting data transmission. In the *Ready* state, the *POC* waits for commands from the *CHI*, and enters to *Wakeup*, *Startup* or *Config* state.

- In the *Wakeup* state, the *POC* tries to send a wakeup signal to other nodes in the cluster for activating their communication controller.

- In the *Startup* state, the *POC* performs the initial synchronization of all nodes.

- After the startup process, the *POC* enters the *Normal Active* state. In the *Normal Active* state, there is no error at this node, or the node has few errors but it can maintain normal data transmission.

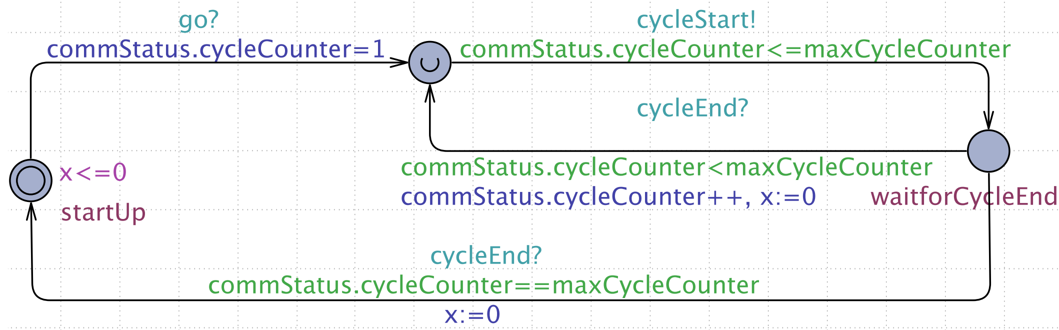


Figure 3.11: POC automaton.

- The *Normal Passive* state indicates that access collisions occurs because of deviation of synchronization so that frames cannot be transmitted.
- If there is a rigorous error, or some errors that keep coming up, the *POC* moves to the *Halt* state. If errors are fixed, the *POC* back to the *Normal Active* state.

In the FlexRay system, a node is initialized according to system configuration firstly. Some nodes in power-saving mode can be awakened by a wakeup signal. Prior to data transmission, all nodes have to be synchronized through sending a synchronous frame in the communication cycle. Next, the nodes start frame transmission in the *Normal Active* state. The frame transmission process is mainly performed by MAC and FSP.

We establish the *POC* automation in the *Normal Active* state for starting transmission, as shown in Fig. 3.11. Because, this work is aimed to verify timed properties of communication, and the processes of the system launching is ingnored. We define some channels to synchronize clock between nodes in UPPAAL.

In the *Normal Active* status, the node is ready to transmit frames during communication cycle. The *POC* is going to start communication cycle, which immediately leaves the initial location `startUp` with a update (`commStatus.cycleCounter:=1`) to a urgent location. The *POC* checks whether the value of the cycle counter is less than or equal to the maximum number of cycles. If the constraint is satisfied, the *POC* uses the channel `cycleStart` to *StaticMAC* and start a communication cycle. Then, the *POC* waits for the cycle end in the location `waitforCycleEnd` until it detects a synchronization channel `cycleEnd`. It will recheck the value of the cycle counter. If the value is less than the maximum value of the cycle counter, the *POC* will update the value plus one and reset the clock `x` for starting next communication cycle. If the value is equal to `maxCycleCounter`, the *POC* will return to the initial state, and reset the clock.

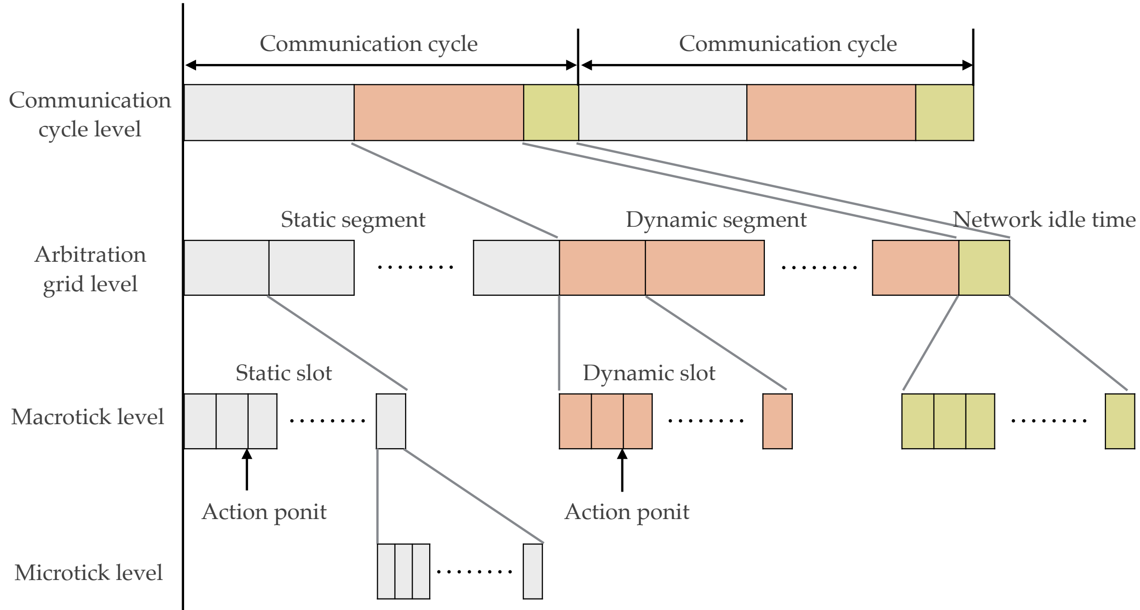


Figure 3.12: Communication cycle of FlexRay model

3.3.3 Media Access Control

The MAC manages channel access and competition between nodes in the cluster. The FlexRay protocol realizes the MAC based on the TDMA scheme, and its communication takes place in a period communication cycle. The communication cycle is divided into four hierarchies, communication level, arbitration grid level, macrotick level and microtick level, as shown in Fig. 3.12.

- In the communication cycle level, a cycle is composed of static segment, dynamic segment, symbol window and NIT.
  - The static segment uses TDMA to access FlexRay bus based on time-triggered, and is composed of static slots. The length of the static segment is fixed.
  - The dynamic segment uses the minislot-based TDMA to transmit message, and it is comprised of a fixed quantity of minislots. The length of the dynamic segment is dynamically variable corresponding to dynamic frames transmitted.
  - The symbol window is used to synchronize communication cycle between nodes by sending a symbol in the communication cycle. The length of symbol window is configurable by macroticks. If the symbol window is not required, it is possible to configure zero maroticks. In this research, the IVN system is

assumed to be synchronous, so the symbol window is unnecessary and it does not influence the timed property of the IVN system.

  - The NIT is a period of free time, which contains all the macroticks which are not utilized in the previous three segments.

- The arbitration grid level identifies the time interval for the message transmission, and specifies which message is being transmitted. It is critical for MAC process. The static segment is made up of static slots of the same length, and the dynamic segment is made up of mini-slots with smaller length.

  - The static slots have same length and they are signed by numbers in ascending order from the beginning of a communication cycle. These static slots are assigned to different nodes that are in the same cluster. The allocation of the static slots is unchangeable during system running. The assigned static slots only transmit messages from specific nodes. If there is no data assigned in a static slot, the FlexRay bus will remain idle until the next static slot comes.
  - Dynamic slots are allocated to transmit dynamic messages in the FlexRay system. The time duration of dynamic slots is determined by the length of message being transmitted, and is composed of minislots. When all of nodes have received the message, next slot can be started. If there is no messages to be sent in a dynamic slot, all nodes have to wait a mini-slot. If the remaining minislots in the dynamic segment are not enough for transmitting next message, the message will wait for next communication cycle. When the dynamic segment ends, remaining minislots will be NIT.

- The macrotick is composed of several microticks, and multiple macroticks constitute a static slot, or a mini-slot. Each static slot and mini-slot have an action point to be offset.

- The microtick is the minimum time unit and is used to guarantee the overall clock synchronization. The length of microtick is determined by clocks of the FlexRay communication controller.

We consider the timing hierarchy from the communication cycle level to the macrotick level in FlexRay model. The communication cycle is segmented into some slots with their identifier in order, and the slots are allocated to nodes. Therefore, a frame of the node has to be transmitted during the corresponding slot with the same identifier. The MAC controls each segment of the communication cycle and the static segment and the dynamic

segment employ different communication schemes. Fig. 3.13 shows the MAC process in specification and description language (SDL).

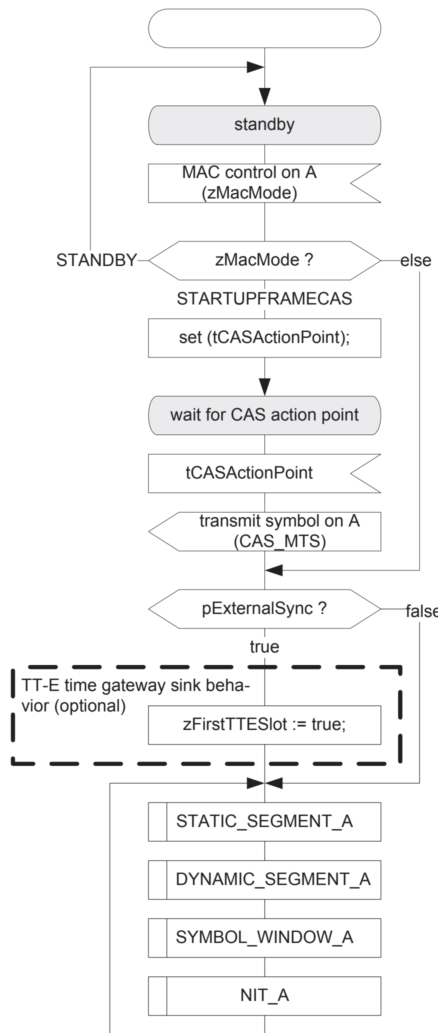


Figure 3.13: Media access process

After the *POC* executed the startup, the *MAC* receives a `STANDBY` command from the *POC* and waits a `zMacMode` signal. If the `zMacMode` indicates `STANDY`, the MAC will reset parameters related to communication cycle, such as action point of slots, the length of slots and minislots, and return to the `STANDBY`. The `zMacMode` is equal to `STARTUPFRAMECAS` which indicates that the collision avoidance action point need to be reset, and the node will send a frame on channels. The *MAC* inspects whether the node is synchronous with other nodes. If the `zMacMode` is else, the MAC will skip the collision avoidance operation and check clock synchronization. The `pExternalSync` indicates that the node is externally

synchronized. If it is true, the `zFirstTTESlot` will be set as true. If it is false, the *MAC* will jump to execute the communication cycle repeatly, including static segment, dynamic segment, symbol window and NIT.

Before performing the communication cycle, the MAC generates the consistency of parameters and clock synchronization. These operations can be realized by the *Configuration* module and synchronization channels in UPPAAL. Thus, the *POC* model directly starts a communication cycle from the static segment.

Static Media Access Control

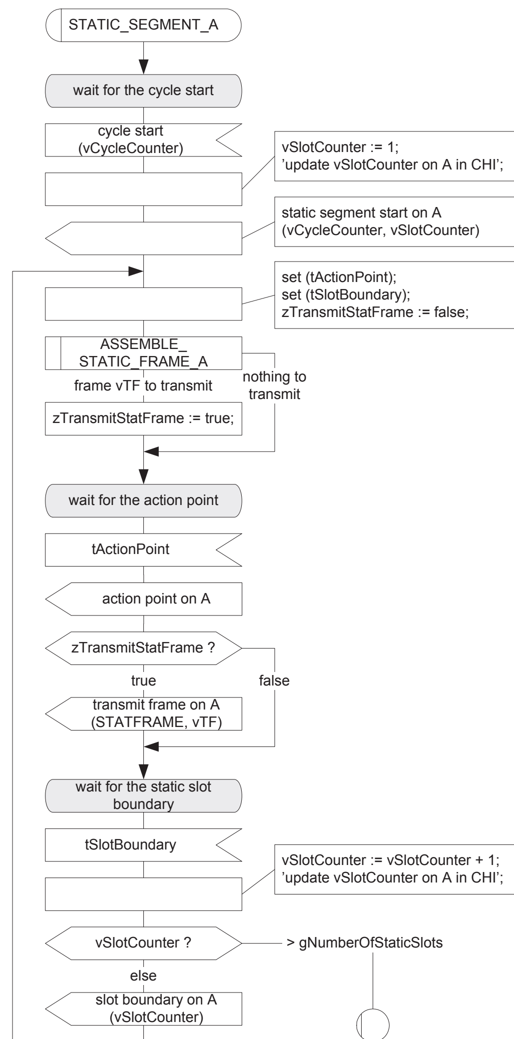


Figure 3.14: Media access in static segment

In the static segment, the *MAC* uses static cyclic scheduling (SCS) to transmit static

messages. Transmitting a static message is depended on the schedule table which is defined by the *Configuration*. Fig. 3.14 shows media access control in the static segment by SDL.

At first, the *StaticMAC* waits for the communication cycle start. While the *POC* starts the communication cycle, the cycle counter starts counting. Meantime, the value of the slot counter is updated to 1, and the first static slot is started. The *StaticMAC* informs the bus that the values of the cycle counter and slot counter, and sets the action point of slots and the length of slot corresponding to the configuration, and then updates *zTransmitStatFrame* to false. If there is a data need to be sent, it will be assembled to a FlexRay frame, and the *StaticMAC* will go to wait for the action point. Then, the *StaticMAC* checks *zTransmitStatFrame*. If the value of *zTransmitStatFrame* is true, the frame will be transmitted. Otherwise the *StaticMAC* will turn to wait for the current static slot ending. At the edge of this slot, the slot counter will be increased. Finally, the *StaticMAC* checks the value of the slot counter. If the value is less than the maximum value of the static slot number, the *StaticMAC* goes back to start next slot; if it is greater, the static segment ends and dynamic segment should be started.

The *StaticMAC* automaton represents how the FlexRay model transmitted messages in the static segment (see Fig. 3.15). When the *POC* starts a communication cycle, the *MAC* performs the static segment. The *StaticMAC* leaves the initial location `waitforCycleStart` and slot counter starts counting from 1. In the static segment, there are some static slots with the same length and action point. The location `waitforActionPoint` with a time constraint (`x<=slotActionPoint`) is waiting for offset of clock synchronization. After the action point, the *StaticMAC* tries to send messages in the location `sendMsg`. If there is a message waiting for transmission in this slot, the *StaticMAC* checks the validity of the message, transmits it as a frame in the slot with the same identifier as the message using the function `transmit()`, and update clock x and `busStatus` as `TRANS`. Then, the *StaticMAC* enters location `waitforTransmissionEnd`. When the frame transmission finished, that is `x==busFlexRay.length`, the bus status will be changed to `CHIRP`. If no messages are waiting for transmission, the *StaticMAC* goes to another location `waitforStaticSlotBoundary` to finish this slot. When the clock x is equal to `staticSlotLength`, the bus status will be labeled as `IDLE`. While the slot ends, the *StaticMAC* inspects the value of the slot counter. If the slot counter is less than the maximum number of static slots, `commStatus.slotCounter<numberOfStaticSlots`, the slot counter plus one, and next static slot starts. When the `slotCounter` reaches the maximum number of static slots, the static segment is completed and the *MAC* executes to the dynamic segment or goes to the NIT segment.

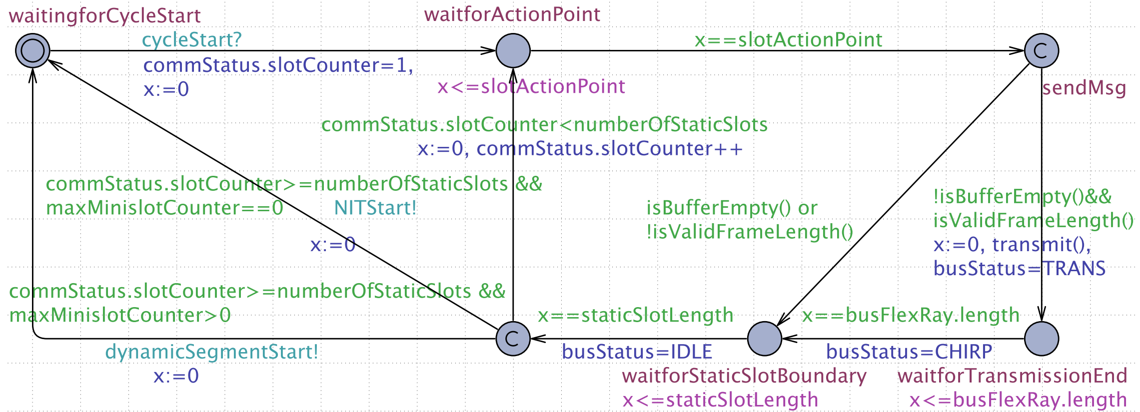


Figure 3.15: StaticMAC automaton.

Dynamic Media Access Control

In the dynamic segment, the MAC uses fixed priority scheduling (FPS) to transmit dynamic messages. The dynamic message has an identifier to denote its priority. Fig. 3.16 shows media access in the dynamic segment by SDL.

The process of the dynamic segment is more complicated than the static segment. Firstly, the MAC checks if any minislots are assigned for the communication cycle. If the number of minislots is zero, there is no dynamic segment; if not, the dynamic segment will be started and hold the slot counter. At the beginning of dynamic slots, the action point of mimislots has to be compared with the action point of slots. Fig. 3.17 shows the action point in two cases. If the duration of offset in the minislots is longer than that in the slots, the action point of minislots should be set to the action point of the first dynamic slot. Otherwise, the duration of offset in the first dynamic slot is equal to the offset time of the slots plus the offset time of the minislots. Note that this is only for the first dynamic slot. In other cases, the action point of dynamic slots is equal to the action point of the minislots. Then, the MAC updates parameters about communication, such as the current number of minislots. The *DYBN_SEG_LOOP_A* starts each dynamic slot. At the end of the dynamic segment, the data in the CHI will be updated.

The dynamic segment loop controls transmissions of each dynamic slot and this process is introduced in Fig. 3.18. Before the transmission begins, the MAC checks to see if there is enough minislots left for transmission, or if there is any slot counters desynchronized to cause the transmission is not allowed in this dynamic slot.

If a transmission is allowed, the *DynamicMAC* assembles the dynamic frame and transmits the frame. If the frame is empty and the bus is idle, this node waits for the transmission of other nodes until the end of the mini-slot. If there is no transmission in this

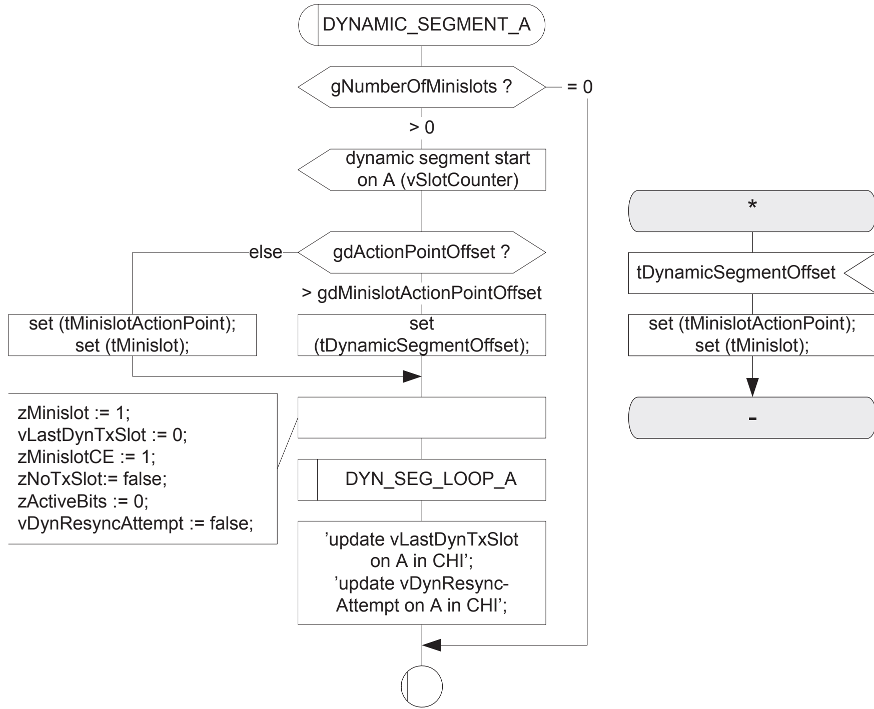


Figure 3.16: Media access in dynamic segment

dynamic slot for all nodes in the cluster, the dynamic slot will be finished with the end of the mini-slot. If there is a *CE start* signal on the channel, that is, there is a node that has a frame to send in this dynamic slot. All the other nodes in the cluster will wait for the end of its transmission and update the value of parameters.

While waiting for the frame transmission to end, if a `detection trailing sequence (DTS)` signal is received by a node, the node will lock the end of the dynamic slot for avoiding potential noise affect the length of the dynamic slot. If a `potential idle start` signal is received before a `CHIRP` signal, it means that the frame transmission ends in the current mini-slot, and the mini-slot is the last one of the dynamic slot. During the frame transmission, the value of mini-slot counter has to be synchronized. If the current mini-slot is the last of the dynamic slot and there is no more message to be transmitted, the value of the mini-slot counter should increment as well as the value of the slot counter, and the MAC exits the dynamic segment loop and is going to wait for the end of the dynamic segment. When the value of the slot counter is needed to increment, if the value is over the maximum number of slots, the loop will be exited, if not, the loop will be continued and goes back to wait for the end of this slot.

When a `CHIRP` signal is detected by the MAC, the nodes go into wait for the end of the

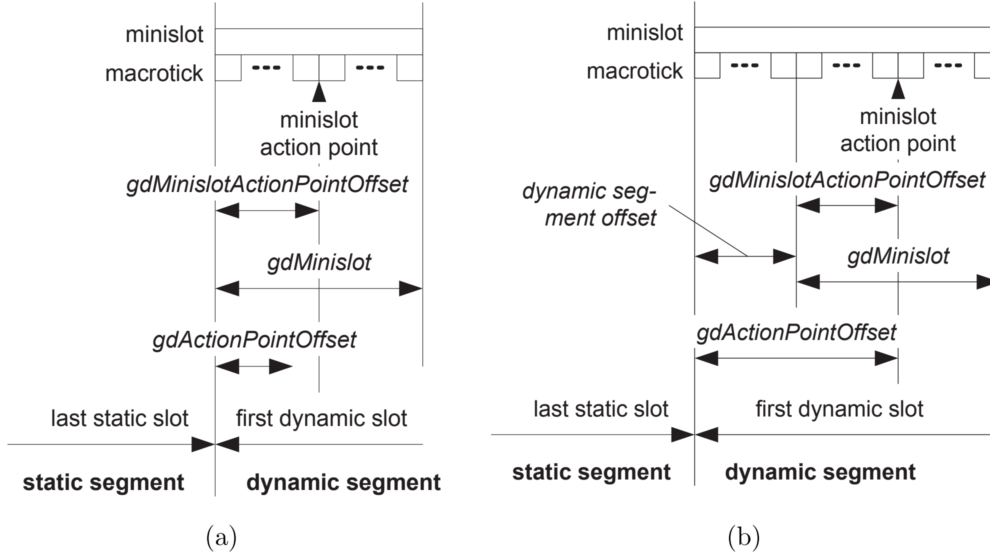


Figure 3.17: (a) action point a (b) action point b

dynamic slot. If a `CE start` signal appears on the bus, generally, it indicates there is a fault that caused by noise. The nodes arrange the new transmission and correct the value of the mini-slot and slot counters. Under normal communication, no `CE start` signal is received at that waiting state. While the present value of the mini-slot counter is equal to the last mini-slot number of the dynamic slot, the dynamic slot is end and the value of the slot counter is increased. If the frame threshold is longer than the length of the frame, a new slot will be added with the length of one or two mini-slots and the *DynamicMAC* resynchronizes between nodes. Then, the *DynamicMAC* exits the loop with the maximum number of slot, or continues the loop and sets the mini-slot action point again.

After the *StaticMAC*, the *DynamicMAC* automaton starts with a channel `dynamicStart`, as shown in Fig. 3.19, if the dynamic segment was defined, the number of mini-slots is greater than 0. The *DynamicMAC* has two counter, the slot counter will keep counting from the `maxStaticSlots` plus 1, and the mimi-slot counter starts from 0. According to the SDL description, the *DynamicMAC* compares slot action point and mini-slot action point first, and the bigger one will be the dynamic slot action point. The *DynamicMAC* waits for time to pass at the location `waitforSlotActionPoint` until the clock `x` is equal to `dynamicSlotActionPoint`, and starts next mini-slot. Note that the remaining time duration of the first mini-slot of each dynamic slot, will be complemented in the frame transmission process, the location `waitforTansmissionEnd`. Before frame transmitting, the *DynamicMAC* also checks whether there is a message waiting to be transmitted in this dynamic slot, and whether there is enough mini-slots for this message. If the

Figure 3.18: Media access in dynamic segment arbitration

constraint is satisfied, the bus status will be changed to `TRANS`, and the *DynamicMAC* transmits the message to bus using the function `transmit()` and moves to the location `waitforTransmissionEnd`. When the transmission is complete, The *DynamicMAC* will recalculate the number of mini-slots and update the mini-slot counter according to the value of the current clock `x`. At the same time, the bus status is updated to `CHIRP`. When the clock `x` satisfies this dynamic slot boundary the dynamic slot ends and the bus becomes idle status (`busStatus=IDLE`). If there is no messages waiting to be transmitted in this dynamic slot, or there is no enough mini-slots for this message, the length of the dynamic slot is a mini-slot. The *DynamicMAC* goes to the location `waitforMinislotEnd`, and it leaves this location when the clock `x` is equal to `minislotLength` as well as the

Figure 3.19: Dynamic MAC automaton

bus status becomes `IDLE`.

While a dynamic slot is finished, the *DynamicMAC* examines the values of the slot counter and minislot counter.

- If the value of the slot counter is less than the maximum number of slots, the *DynamicMAC* will perform next dynamic slot from setting the dynamic action point. But there is no more mini-slots for frame transmission, that is the value of the mini-slot counter is `maxMinislotCounter`, the *DynamicMAC* will starts *NIT* segment and reset the mini-slot counter.

- If the value of the slot counter is equal to the `maxSlotCounter` and mini-slots have just been used up (`commStatus.minislotCounter==maxMinislotCounter`), the dynamic segment ends and the *MAC* performs the *NIT* segment. If there are still some mini-slots (`commStatus.minislotCounter < maxMinislotCounter`), *DynamicMAC* will consume these mini-slots and then starts the *NIT* segment.

Network Idle Time

The *NIT* is a segment of network idle time after the static and dynamic segments, which contains all macroticks that are not used in the previous. If there is no remaining macroticks, the *NIT* still has two macroticks at least. Fig. 3.20 shows the processing

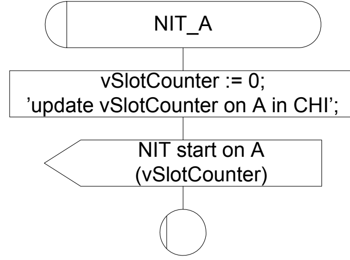


Figure 3.20: Architecture of the IVN system design.

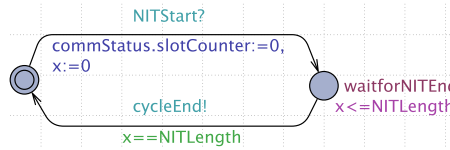


Figure 3.21: NIT automaton

of NIT in SDL. During the *NIT*, all of nodes in the cluster need to reset the value of the slot counter to zero, and elapse the surplus time in the communication cycle. When the *NIT* ends, the *POC* will start next communication cycle from the static segment.

The *NIT* automaton is shown in Fig. 3.21. The *NIT* is started with a channel `NITStart` from *DynamicMAC*, while the value of the slot counter is reseted to 0. Since the *DynamicMAC* already spent the remaining mini-slots in the communication cycle, the *NIT* only keeps the minimum length of the NIT segment. When the *NIT* ends (`x<=NITLength`), the *NIT* will communicate with the *POC* using the channel `cycleEnd`, In other words, the current communication cycle ends and the *POC* turns into next communication cycle.

3.3.4 Frame and Symbol Processing

The *FSP* is used to receive messages during each slot. As soon as a transmission begins, the bus state is identified as active. While the transmission finishes, the slot ends, the bus state is updated to idle. The FSP monitors the state of the bus in real time and executes receiving function. Fig. 3.22 shows the state transition of *FSP*. In the *standby*, the execution of the *FSP* is halted. If there is a *CE start* signal detected, the *FSP* starts frame decoding and then goes to wait a *CHIRP* signal. The *CHIRP* denotes that the frame has been completely transmitted and the bus idle recognition point is marked. When the *FSP* detects a *CHIRP* signal, it will wait for the end of the transmission. The receiving process starts by detecting a*CE start* signal and ends by detecting a *CHIRP* signal. The frame decoding process does not affect the time of receiving message. So, we

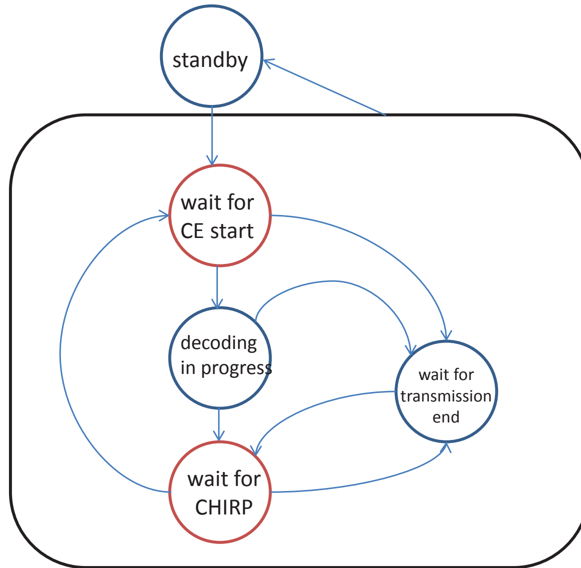


Figure 3.22: Overview of frame and symbol processing

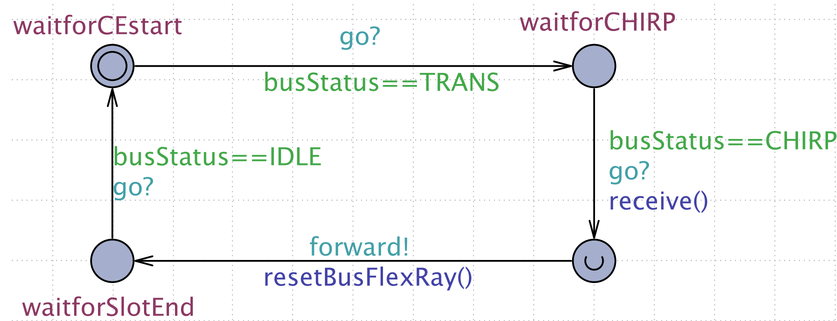


Figure 3.23: FSP automaton

abstract other three states to present the frame reception as shown in Fig. 3.23.

In the initial location `waitforCEstart`, the *FSP* is waiting for a tramsmission. When the bus status becomes to `TRANS`, there is a message transmitting. Then, the *FSP* is waiting for the bus status changed to `CHIRP`. When the message transmission finished, the *FSP* executes the function `receive()` to store the message to FlexRay interface, and checks whether the indentifier of the message is matched with the current slot number. The *FSP* will notify receivers that the message has been received by channel `forward`, and reset FlexRay bus. When the current slot ends (`busStatus==IDLE`), the *FSP* turns into the initial location and waits next message transmission.

This chapter presented a strategy to abstract IVN systems. The first stage is to abstract

the architecture of IVN systems and simplify the topology of the system. The second stage is to abstract the composition of the protocol specifications and identify the necessary functions from the specifications of CAN and FlexRay. We illustrated the abstraction and construction process of two communication control models in UPPAAL. We will abbreviate these two models as CAN model and FlexRay model. They will be the core of the IVN system model, and each bus will contain a protocol model for communication. That is the two models are reusable in every IVN system model. Based on this and the layer structure of the specifications, we will present a framework to construct IVN system models.

Chapter 4

A Framework for Modeling IVN Systems

In this chapter, we will propose a framework to model IVN systems in UPPAAL, and explain how to use the framework to model an IVN system in the design phase. Firstly, a UML class diagram is provided to show the relations between each modules in the framework, and the functions and parameters of these modules. Since some modules are composed of multiple automata, we use a component diagram to represent the relationship between automata. The *Communication controller* of the framework, which consists of the CAN model and FlexRay model, is introduced in pervious chapter. Then we give the relevant *Configuration*, *Interface*, *Medium*, *Environment* and *Gateway* modules. At last, we will show how to use the framework to construct an IVN system design model and discuss the reusability of the framework.

4.1 A Framework in UPPAAL

Abstracted IVN systems may employ different numbers of environments, these environments may use different types of buses to connect gateways in various topologies. The diversity of IVN systems means that we need a reusable framework to model and verify communication behaviors with both CAN and FlexRay protocols. Referring to the protocol structure we presented in the previous chapter, we propose a reusable framework, including the *UPPAAL* engine, *Transmission medium*, *Communication controller*, *Interface*, *Environment*, *Forwarder* and *Configuration* modules, as shown in Fig. 4.1.

- The *UPPAAL* module is the foundation on which the other parts are built. We model the other modules, and simulate and check IVN system design models in the

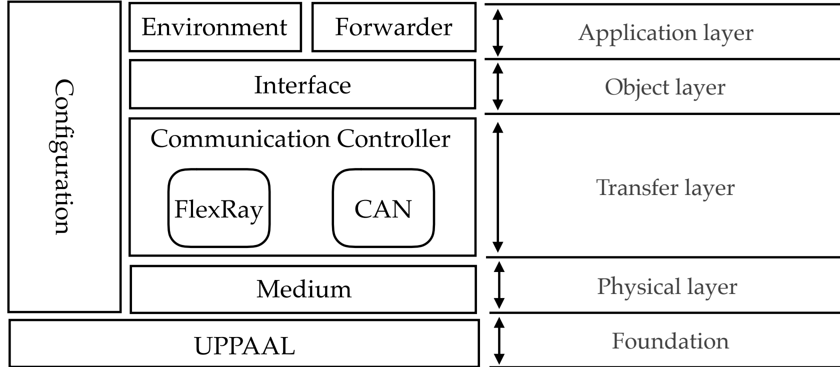


Figure 4.1: The framework.

UPPAAL.

- The *Medium* module simulates message transmission over physical wires as the *Physical layer* of the specifications.
- The *Communication controller* holds CAN model and FlexRay model describing communication behaviors based on the *Transfer layer* of the specifications.
- The *Interface* module is including buffers designated for specific messages, and operates as a communication bridge between the *Application layer* and *Transfer layer*. This module represents the *Object layer* in the CAN specification and the *Presentation layer* in the FlexRay specification.
- The *Environment* module is task models that write/read message to/from the *Interface*, as the *Application layer* of the specifications.
- The *Forwarder* module also belongs to the *Application layer* and is a special task to communicate between different protocols. It executes frame-forwarding processes, representing an essential component to convert protocols.
- The *Configuration* module contains parameters related to the other modules. In the proposed framework, the *Bus*, *Communication controller*, *Interface*, and *Configuration* modules are fixed and reusable in terms of the environments and gateways; the *Environment* and *forwarder* modules are changeable in terms of sending and forwarding messages in different ways.

The framework can be used to construct a design model of an IVN system. We use a class diagram of the framework to display the relationships between modules, and all

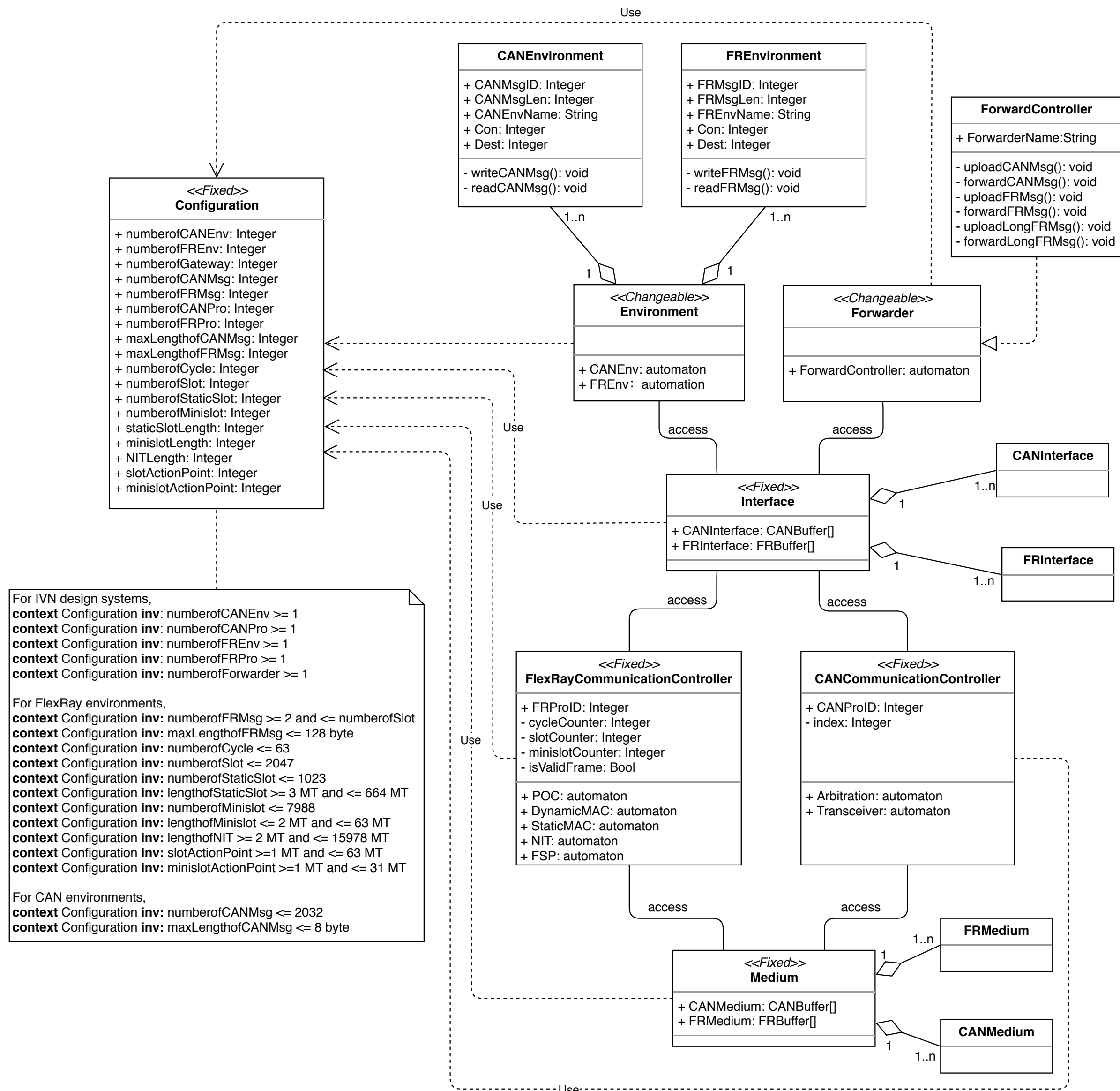


Figure 4.2: The class diagram of the framework in UML.

attributes and operations, as shown in Fig. 4.2. To reflect the feature of the modules, we define two special stereotypes, *<<Changeable>>* and *<<Fixed>>*. The *<<Changeable>>* means that the class can be changed according to the system design. The *<<Fixed>>* means that the class is fixed and reusable in terms of modeling different IVN systems. We will describe the UML class diagram according to the modules of the framework.

- The *Environment* class:
  - The *Environment* class is presented by a *<<Changeable>>* class. Generally,

each node may has different tasks to perform various operations and processing on data in the IVN system. Although we can ignore the concrete functionalities of applications in ECUs, the possible tasks are still infinite. Thus, the *Environment* model could be changed by designers.

- The *Environment* class is aggregated by *CANEnv* and *FREnv* classes.Since an IVN system consists of CAN environments and FlexRay environments that need to conform to the bus protocols to send messages, we give two subclasses for CAN environments and FlexRay environments. The *CANEnv* and *FREnv* are described by automata in UPPAAL.
- Although the *CANEnv* and *FREnv* can perform various operations and processing on data, it will eventually need to send and receive messages. Thus, sending and receiving messages are essential functions for an environment. The *CANEnv* and *FREnv* contains two primary functions for delivering and reading messages, such as the `writeCANMsg()` and `readCANMsg()` in the *CANEnv*. To send a message, the writing function writes the message to a corresponding buffer in the *Interface* as a frame, and then the *Protocol* controls its transmission. To receive a message, the reading function reads the message from the *Interface*. Also, each environment need to set the identifier, length of messages it sends (e.g. `CANMsgID` and `CANMsgLen`)), the identifier of the bus to which it is connected (e.g. `ConBus`), the identifier of the bus the message needs to reach (e.g. `DestBus`) and its own name (e.g. `CANEnvName`) in the system. Note that `CANMsgID` and `FRMsgID` can not be the same. Because the CAN message/FlexRay message will be sent to the FlexRay bus/ CAN bus.

- The *Forwarder* class:
  - The *Forwarder* module is also a ***<<Changeable>>*** class, and can be realized by a *ForwardController* automaton depending on the IVN system design. The main function of the *ForwardController* is to forward messages between the *CAN* and *FlexRay*. In reality, gateways have different features, such as scheduling algorithms, which may affect the time property of the IVN system. Thus, the *Forwarder* module can be changed by different automata.
  - The process of forwarding messages is executed by two functions for each protocol. The uploading functions, `uploadCANMsg()` and `uploadFRMsg()`, read a message from the *Interface* and upload the message to a temporary buffer. The forwarding functions, `forwardCANMsg()` and `forwardFRMsg()`, transform

the format of the message and relay the frame to a corresponding buffer in the *Interface*. In particular, since the payload length of CAN frame is less than that of FlexRay frame, if the length of FlexRay message is greater than the maximum length of CAN message, a `uploadLongFRMsg()` function will divide it into several CAN messages. Subsequently, these CAN messages will be released in order by the `forwardLongFRMsg()` function. .

- The *Interface* class:

  Interface module is defined by a *Interface* class with ***<<Fixed>>***, and aggregates the *CANInterface* class and *FRInterface* class. The *Interface* provides buffers for storing messages exchanged between the *Application layer* and *Transfer layer*. Although CAN messages and FlexRay messages have different frame format, not all data fields of the frame are necessary in our model, for example, cyclic redundancy check (CRC) field is used to guarantee the correctness of the frame. Because our model is an abstraction for an error-free communication system, the data field with regard to error checking is ignored. An abstracted frame format is proposed for both CAN messages and FlexRay messages, including frame identifier, payload length, environment identifier and destination identifier. Based on the frame formats, the *CANInterface* and *FRInterface* are constructed by the buffers. We define two types of buffers, sending buffers for putting in messages that need to be sent and receiving buffers for storing received message.

- The *CAN* class and *FlexRay* class:

  The *Communication Controller* is the kernel for transmitting messages, and performs communication behaviors following specifications. We separately define the *CAN* class and *FlexRay* class as ***<<Fixed>>***, because they have their own special communication mechanism. We build fixed *CAN* and *FlexRay* models in UPPAAL, and these can be reused to construct different IVN systems.

  – The *CAN* class has two attributes, `CANBusID` and `index`. The `CANBusID` is a unique identifier for a bus. The `index` is a variable in automaton *Arbitration*. The *CAN* model is composed of two automata, *Arbitration* and *Transceiver*. The CAN communication controller transfers messages based on the static priority scheduling algorithm. The *Arbitration* monitors the sending `CANBuffer[]` in real time, and reads a message with the highest priority. The *Transceiver* is responsible for delivering the message to *CANMedium*, and stores the message to the corresponding receiving *CANInterface* when the transmission ends.

- The *FlexRay* class has some attributes. The `FRBusID` is a unique identifier for a bus. Others are used to automata. The *FlexRay* model transmits messages in determinable time slots of communication cycle, which has several automata to serve communication cycles refer to the FlexRay specification. The protocol operation controller (*POC*) monitors the beginning and ending of a communication cycle, and counts the number of communication cycles. The *StaticMAC*, *DynamicMAC*, and NIT simulate the communication cycle and arbitrate messages. The *FSP* monitors the bus state and transmit or receive messages from the *FRInterface*. Here, all automata are synchronous to ensure the consistency of communication cycle.

- The *Medium* class:

  The *Medium* class is *<<Fixed>>*, and defines two kinds of hardware wires, *CANMedium* class and *FRMedium* class. They are represented by two buffers that correspond to frame structures. The *CANMedium* is a buffer with the CAN frame format, and the *FRMedium* is a buffer with the FlexRay frame format.

- *Configuration* class:

  The *Configuration* class holds all parameters in the *Environmnet*, *Forwarder*, *Interface*, *CAN*, *FlexRay* and *Medium*. We describe them and give their range of values, as listed in Table 4.1. To build an IVN system model, we need to specify the number of CAN and FlexRay environments, buses and forwarders, and then set up each message and bus protocol. No.1 to 5 describe components of an IVN system; No.6 to 16 describe FlexRay messages and communication cycle of FlexRay environments; No.17 and 18 clarify CAN messages. These parameters are described using the object constraint language following the protocol specifications and constraints, as shown in Fig. 4.2. We use the macrotick (MT) (the smallest time unit in the communication cycle) to represent the execution time of messages on the bus. Setting *Configuration* is necessary to model an IVN system design model.

As the class diagram described, some modules are created in the form of automata in UPPAAL. We drew a component diagram against Fig. 4.2 to show relationships between automata, as shown in Fig. 4.3.

- The *Environment* has two types of task automata, *CANEnv* and *FREnv*. They deliver/receive messages to/from interfaces the *Interface* provided. The *CANEnv* is only allowed to access *CANInterface*, and the *FREnv* is only allowed to access

Table 4.1: Parameters in the *Configuration*

No.	Name	Description	Range
1	numberofCANEnv	Number of CAN environments	>= 1
2	numberofFREnv	Number of FlexRay environments	>= 1
3	numberofForwarder	Number of gateways	>= 1
4	numberofCANBus	Number of CAN buses	>= 1
5	numberofFRBus	Number of FlexRay buses	>= 1
6	numberofFRMsg	Number of FlexRay messages	1 - 2047
7	maxLengthofFRMsg	Maximum payload length of FlexRay messages	0 - 254 bytes
8	numberofCycle	Number of cycles in a given cluster	7-63
9	numberofSlot	Number of slots in a communication cycle	2 - 2047
10	numberofStaticSlot	Number of static slots in the static segment	2 - 1023
11	lengthofStaticSlot	Duration of a static slot	3 - 664 MT
12	numberofMinislot	Number of minislots in the dynamic segment	0 - 7988
13	lengthofMinislot	Duration of a minislot	2 - 63 MT
14	lengthofNIT	Duration of the network idle time	2 - 15978 MT
15	slotActionPoint	Number of macroticks the action point is offset from the beginning of a slot	1 - 63 MT
16	minislotActionpoint	Number of macroticks the minislot action point is offset from the beginning of a minislot	1 - 31 MT
17	numberofCANMsg	Number of CAN messages	0 - 2032
18	maxLengthofCANMsg	Maximum payload length of a CAN frame	0 - 8 bytes

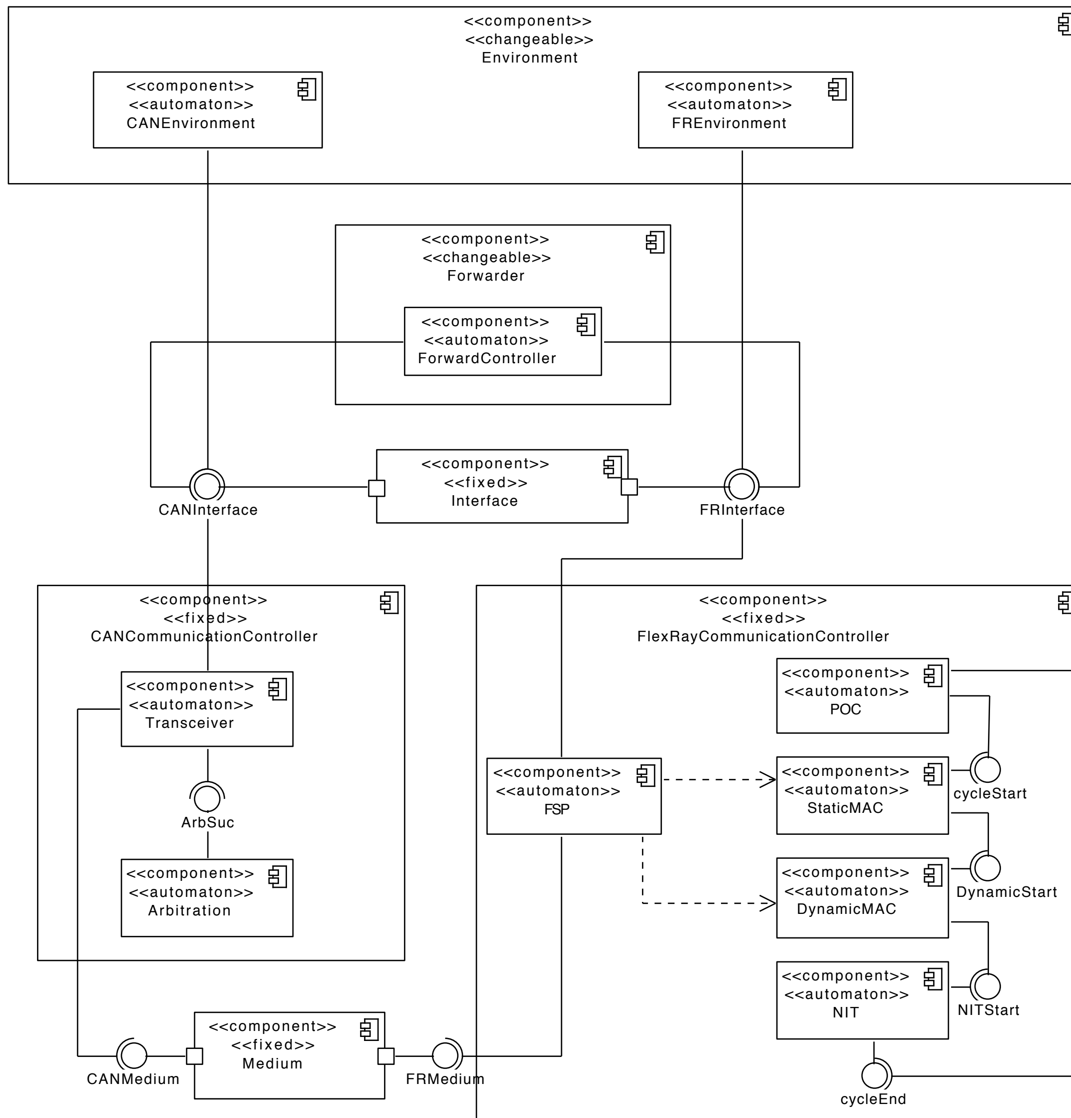


Figure 4.3: The component diagram.

FRInterface. The two types of task automata are independent and work for different environments.

- The *Forwarder* class is implemented by a *ForwardController* automaton to forward messages between CAN environments and FlexRay environments, so it can access the two types of interfaces, the *CANInterface* and *FRInterface*. In addition to receiving and delivering messages, the automaton needs to translate messages between CAN environments and FlexRay environments. Thus, the *Forwarder* class provides several essential operations to accomplish its functions.

- The *Interface* class is defined by two types of buffers, `CANBuffer[]` and `FRBuffer[]`, there is no automata. Several `CANBuffer[]` with different identifiers make up the *CANInterface*, as well as several `FRBuffer[]` with different identifiers make up the *FRInterface*. The *Interface* class provide the two interfaces for exchanging messages between the *Application* or *Forwarder*, and the *Communication controller*.

- The *Communication Controller* has two separate protocol classes, *CAN* and *FlexRay*. We have covered their model in the previous chapter. The *CAN* model and *FlexRay* model are composed of several automata, as listed in the class diagram. The component diagram shows how the automata are related to each other.

  - In the *CAN* class, the *Arbitration* examines messages in the *CANInterface* when the CAN bus is free, and notifies the *Transceiver* which message had won the arbitration by a synchronizing channel `ArbSuc` that is defined in UPPAAL. Then the *Transceiver* accesses the *CANInterface* to get the message and transmits it to the *CANMedium*. When the transmission finished, the message will be stored in the *CANInterface* again by the *Transceiver*, and waits to be received or forwarded.
  - In the *FlexRay* class, four automata, *POC*, *StaticMAC*, *DynamicMAC* and *NIT*, achieve the period communication cycle. They synchronize through four channels, `cycleStart`, `DynamicStart`, `NITStart` and `cycleEnd`. During the communication cycle, the *StaticMAC* and *DynamicMAC* inspect whether any messages need to be transmitted in the *FRInterface* at the beginning of each time slot. If there is, they will change the state of the FlexRay bus, and the *FSP* plays the *Transceiver* of the *CAN* model to access the *FRInterface* and transmits it to the *FRMedium*. When the transmission finished, the message will be stored in the *FRInterface* again by the *FSP*, and waits to be received or forwarded. If no messages need to be sent, they will wait for next time slot.

- The *Medium* class uses two buffers to provide two interfaces, a *CANMedium* and a *FRMedium*. They represent psychical connections between environments and gateways.

In this section, we introduced the framework itself and presented the class diagram and the component diagram to explain the composition of each module in the framework and the relationship between them. Next, following the framework, we will show how modules are implemented in the UPPAAL, except the *Communication Controller* module.

4.2 Configuration Module

An IVN system model is established based on the system design. The *Configuration* describes the necessary information for the system, which contains all of parameters in the IVN system model. These parameters are set according to the abstracted system design, such as the number of CAN and FlexRay environments, messages and buses. We have listed the parameters we need in the class diagram (Fig. 4.2), but how they relate to each automaton. We use a hierarchy diagram to represent them, as show in Fig. 4.4. To describe the abstracted design model of an IVN system, we divide parameters into three categories, stating the components of the system, setting environments in the system, and clarifying CAN protocol and FlexRay protocol. They correspond to each layer of the hierarchy diagram, respectively.

- Components of the abstracted systems

  To construct an IVN system model, we need to state the number of environments, buses and gateways and how they are connected. In Fig. 4.4, the second layer has three parts that corresponds to the compositions of the system. Since environments and buses can meet different protocol standards, we divide environments into CAN environments and FlexRay environments, and divide buses into CAN buses and FlexRay buses, as the third layer of the hierarchy diagram.

  - For CAN environments and FlexRay environments, we declare the number of them (`numberofCANEnv` and `numberofFREnv`).
  - For CAN buses and FlexRay buses, we declare the number of them (`numberofCANBus` and `numberofFRBus`).
  - The *Forwarder* declares the number of forwarders (`numberofForwarder`).

  These parameters state the number of components in the system and do not represent the topology of the system. The topology depends on the setting of variables in specific automata, such as *CANTask*, *FRTask* and *ForwardController*.

- Environments of the abstracted system

  To setting buses, first we need to specify the total number of CAN messages and FlexRay messages in the system (`numberofCANMsg` and `numberofFRMsg`), and the maximum length of them (`maxLengthofCANMsg` and `maxLengthofFRMsg`). These parameters are crucial, since they are related to the *Interface* module and *Communication controller* module. In addition, *CANTask* and *FRTask* are used to

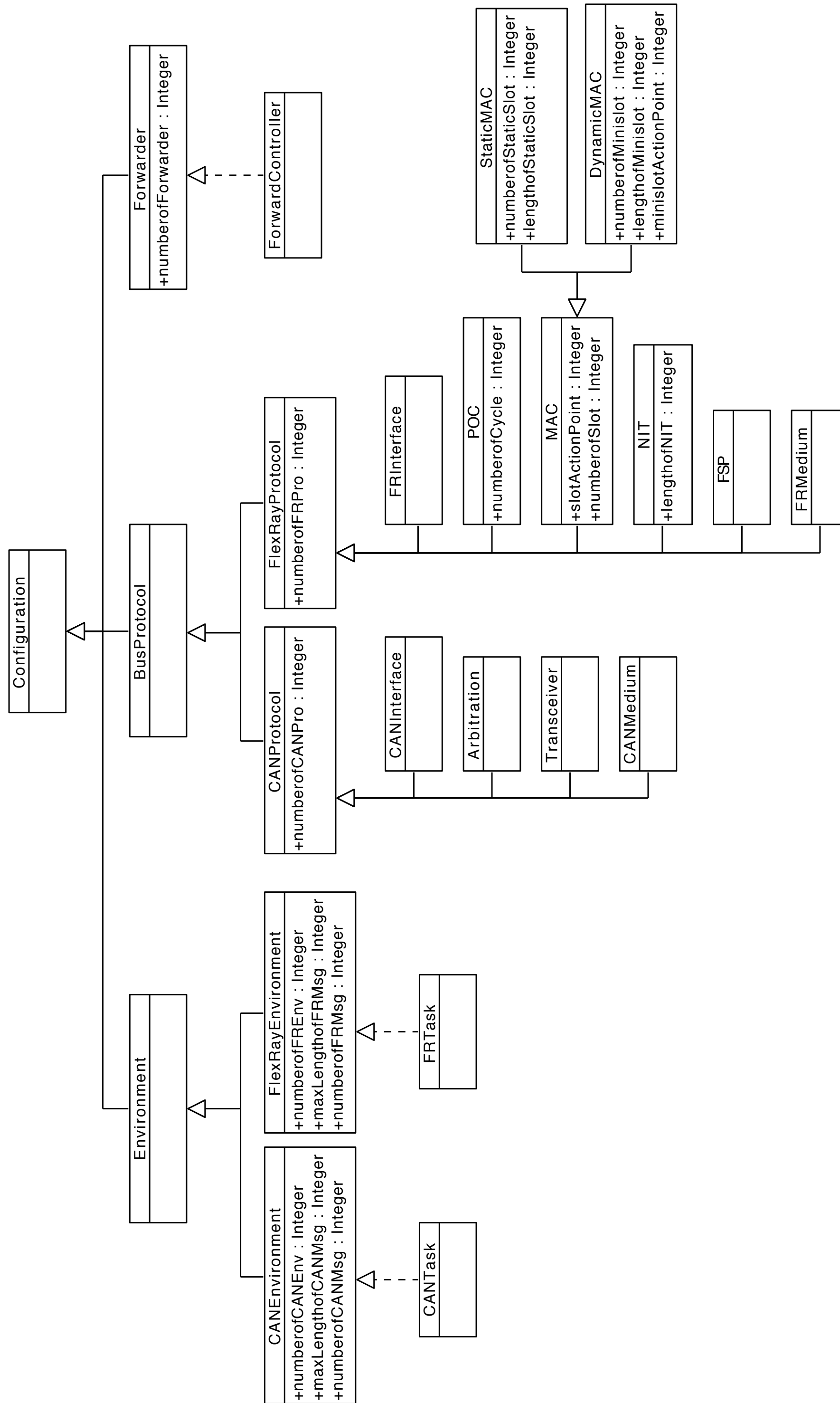


Figure 4.4: The hierarchy diagram of the *Configuration*.

implement environments to release messages to a bus and receive messages from buses. The `numberofCANMsg` and `numberofFRMsg` refers to the number of messages with different identifiers.

- The CAN protocol and FlexRay protocol in the abstracted system

  Between an environment and a forwarder or forwarders, there must be a bus protocol connecting them. The bus protocol conforms to the CAN protocol or FlexRay protocol. The protocol models have some necessary parameters that need to be set, especially FlexRay protocol.

  - The CAN protocol model consists of four parts, none of which have additional parameters. Note that the number of buffers in *CANInterface* be greater than the sum of the `numberofCANMsg` and `numberofFRMsg`. It is possible for each protocol to transmit messages from different environments.
  - Since the FlexRay protocol model sends messages strictly according to a time period, we need to explicitly describe the communication cycle, such as `numberofCycle`, `numberofSlot`, `lengthofSlot` and so on. They are listed in each automaton. The *MAC* represents the media access control process in the communication cycle, including *StaticMAC* and *DynamicMAC*. Also, the number of buffers in *FRInterface* and the number of slots (`numberofSlot`) should be greater than the sum of the `numberofCANMsg` and `numberofFRMsg`.

The *Configuration* module directly affects properties of the IVN system. For example, the number of messages relates to the number of sending and receiving buffers in the interface. If the maximum ID of messages is greater than the number of buffers, data overflow may happen. The length of FlexRay communication cycle influences the response time. If there have not many messages or the length of the communication cycle is too long, the networking idle time will become longer. Hence, we have to set parameters carefully to guarantee reasonability of the system.

4.3 Interface and Medium Module

The *Interface* contains buffers for exchanging messages between the *Environment* or *GatewayController* and *Communication Controller* module. The following code fragment shows frame structures for interfaces and bus mediums in UPPAAL. These buffers use same data structure defined by protocol frames. Each buffer includes four information, the length and identifier of messages (`id` and `length`), the identifier of a protocol connected

to an environment that sent the message (`con`) and the identifier of a protocol connected to an environment that receive the message (`dest`). We set up two kinds of buffers for CAN and FlexRay, sending buffers (`CANMsgForSend[]` and `FRMsgForSend[]`) and receiving buffers (`CANMsgForReceive[]` and `FRMsgForReceive[]`). The sending buffers are used to store messages that need to be sent, and the receiving buffers are used to receive messages that have completed transmission. The number of the CAN buffers are defined by the sum number of CAN messages and FlexRay messages. The *CANMedium* and *FRMedium* are a buffer with the frame structure for simulating physical wire.

```
/* CAN Interface */
typedef struct
{
    CANMsgID id;
    CANMsgLen length;
    ProtocolID con;
    ProtocolID dest;
} CANFrame;
CANFrame CANMsgForSend[numberofCANMsg+numberofFRMsg+1];
CANFrame CANMsgForReceive[numberofCANMsg+numberofFRMsg+1];
CANFrame CANMedium;

/* FlexRay Interface */
typedef struct
{
    FRMsgID id;
    FRMsgLength length;
    ProtocolID con;
    ProtocolID dest;
} FRFrame;
FRFrame FRMsgForSend[numberofCANMsg+numberofFRMsg+1];
FRFrame FRMsgForReceive[numberofCANMsg+numberofFRMsg+1];
FRFrame FRMedium;
```

4.4 Environmnet Module

Each environment has one or more tasks for sending and receiving in the IVN system. These tasks serve as the *Application layer*, which are implemented by *CANEnv* automata and *FREnv* automata. The *CANEnv* and *FREnv* automata execute write-read operation. The following two functions show the write-read operation used in a *FREnv* automaton. The function `writeFRMsg()` writes a message into a sending buffer of the *FRInterface* corresponding `id`, and the function `readFRMsg()` reads messages from the receiving buffer of the FlexRay interface and clears data in the buffer. Similarly, there are two functions, `writeCANMsg()` and `readCANMsg()` are used in *CANEnv* automata and access *CANInterface*.

```
1  void writeFRMsg(FRMsgID id, FRMsgLength length, ProtocolID con,
2       ProtocolID dest)
3  {
4      FRMsgForSend[id].id = id;
5      FRMsgForSend[id].length = length;
6      FRMsgForSend[id].con = con;
7      FRMsgForSend[id].dest = dest;
8  }
9
10 void readFRMsg(FRMsgID id)
11 {
12     FRMsgForReceive[id].id = 0;
13     FRMsgForReceive[id].length = 0;
14     FRMsgForReceive[id].con = 0;
15     FRMsgForReceive[id].dest = 0;
16 }
```

The *Environment* module generates messages and send them to the *Interface* module. Since an IVN system may contain a large number of probabilistic events, we propose two types of *Environment* automata, ordinary automata and probabilistic automata. These two types of *Environment* can all be used to build IVN systems by our framework. Also, we adopt classical model checking and statistical model checking to verify different types of *Environment* in IVN systems.

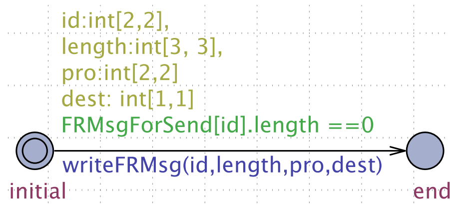
4.4.1 Ordinary Environment Model

The *CANEnv* and *FREnv* automata are built with above functions, and shown as follows. Of course, environment automata are not limited to these types and can be changed by users.

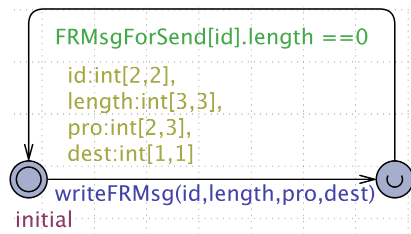
- For the writing operation, the *CANEnv* and *FREnv* may have different time descriptions. Since our target is to verify the timed property of the IVN system, we present three types of tasks, periodic tasks, cyclic tasks, and sporadic tasks in UPPAAL, as shown in Fig. 4.5. In these automata, we use *select* to define the parameters of messages on the state transition, such as `id:int[2,2],length: int[3,3],con:int[2,2],dest:int[1,1]`. There is a difference between the *CANEnv* and *FREnv* automata for writing messages. When the *CANEnv* is writing a message, it has to synchronize with the *Arbitration* of CAN model through a channel `transmissionRequest`.
  - The periodic task writes messages to *Interface* after a fixed time period (`x== Tcycle`).
  - The cyclic task writes messages as soon as the specific sending buffer is empty (`FRMsgForSend[id].length==0`).
  - The sporadic task is executed only once. If and only if the specific sending buffer in the FlexRay Interface is empty, it writes a message using the function `writeFRMsg(id,length,con,dest)`.
- For the reading operation, we employ a simple automaton with one state to perform receiving message, as shown in Fig. 4.6. When the specific receiving buffer in the FlexRay Interface is not empty, the automaton performs the function `readFRMsg(id)`.

4.4.2 Probabilistic Environment Model

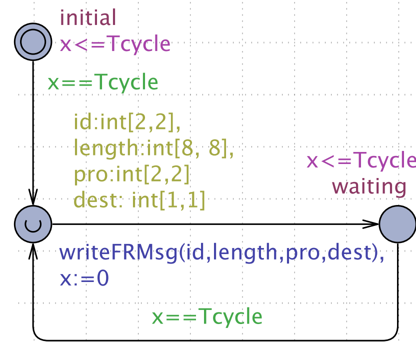
The CAN and FlexRay protocols are deterministic protocols, which has no uncertain behaviors. But the protocols do not define the application layer of nodes, application behaviors are stochastic and non-deterministic. To describe such behaviors, we add probability to the *Environment* module based on the abstraction. There are two ways to realize non-determinacy in UPPAAL. One is the non-deterministic choices between multiple transition paths with probability; the other one is the non-deterministic choices of time delays in a location defined by probability distributions. UPPAAL provides two



(a) The sporadic task



(b) The cyclic task



(c) The periodic task

Figure 4.5: Environment automata for writing messages.

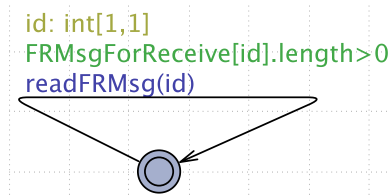


Figure 4.6: Environment automaton for reading messages.

probability distributions, exponential distributions with unbounded delays and uniform distributions with time-bounded delays. We present three kinds of environment models with the stochastic interpretation.

- Probabilistic choices are installed on transitions of stochastic automata. These transitions are from a location to multiple locations with different probabilities. Fig. 4.7 shows an environment automaton with probabilistic choices. Initial location is a urgent state, which is no time elapsed. From the initial location, there are two paths to the `wait_for_trans` location. Each path has a transition in dotted line with a weight, and a time delay with a uniform distribution. The distribution of the overall delay can be obtained by sums of two uniform distribution with weights in Fig. 4.8. To reach the `wait_for_trans` location, one path has a delay time within `3` to `5` with 80% probability; the other path has a delay time within `3` to `10` with

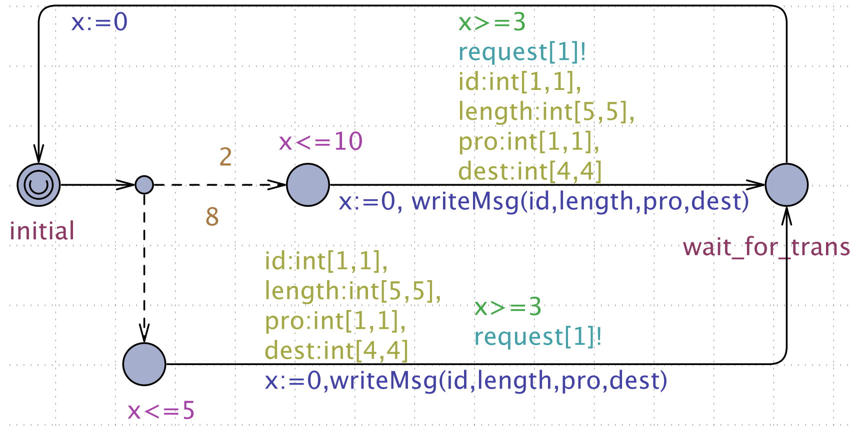


Figure 4.7: Environment automaton with probabilistic choices.

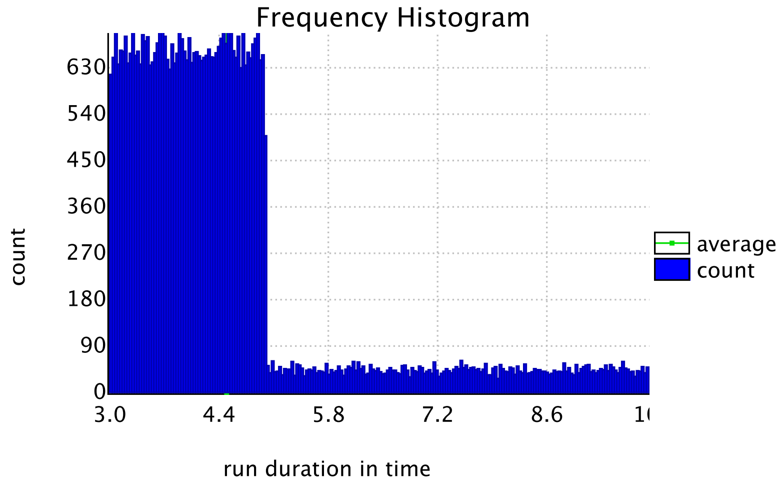


Figure 4.8: Distribution of reachability time with probabilistic choices.

20% probability.

- Uniform distributions are set up to stochastic automata by time constraints. Fig. 4.9 shows an environment automaton with an uniform distribution. The `wait_for_trans` location is used to see the reachability within the time-interval [2,4]. The delay of the transition from the initial state is resolved by an uniform distribution over [2,4]. Hence, the response time is given by the uniform distribution as illustrated in Fig. 4.10.

- Exponential distributions are taken place in locations of stochastic automata. The locations without a time bounded, choose delays according to exponential distributions with rates supplied by users. Fig. 4.11 shows an environment automaton with

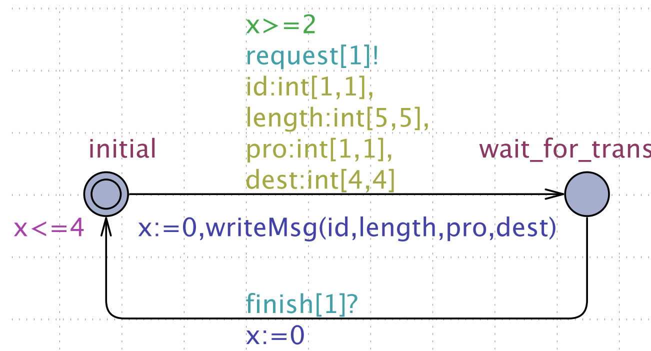


Figure 4.9: Environment automaton with uniform distributions.

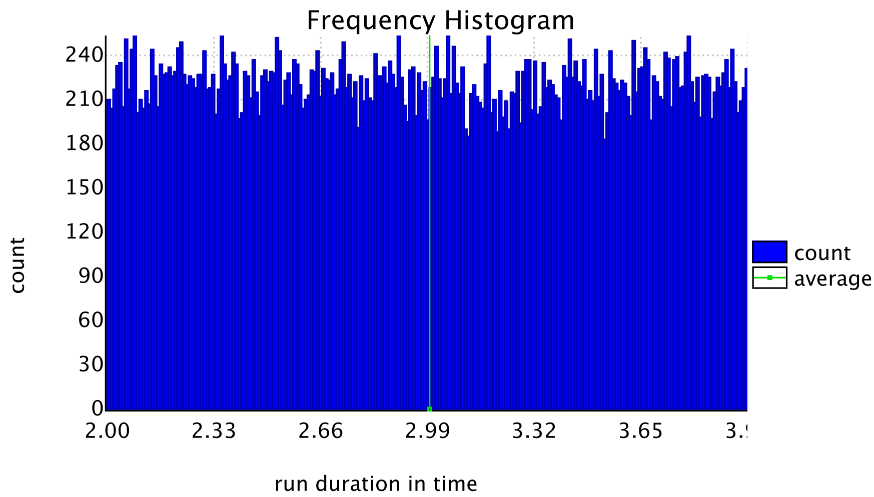


Figure 4.10: Uniform distribution of reachability time.

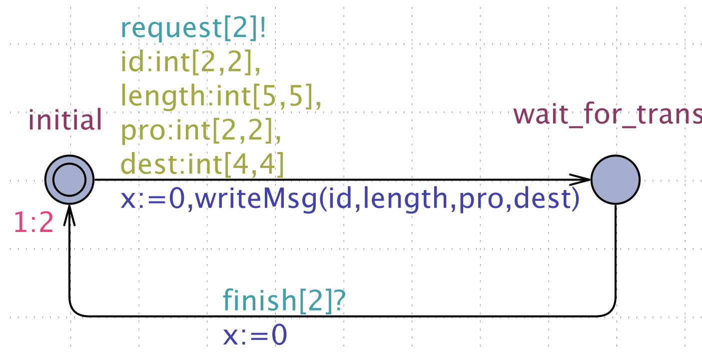


Figure 4.11: Environment automaton with exponential distributions.

an exponential distribution. In initial state, `1:2` is a rate for exponential distribution. After the initial delay, the task will write a message to CAN `Interface`. The resulting distribution of the reachability time is given in Fig. 4.12.

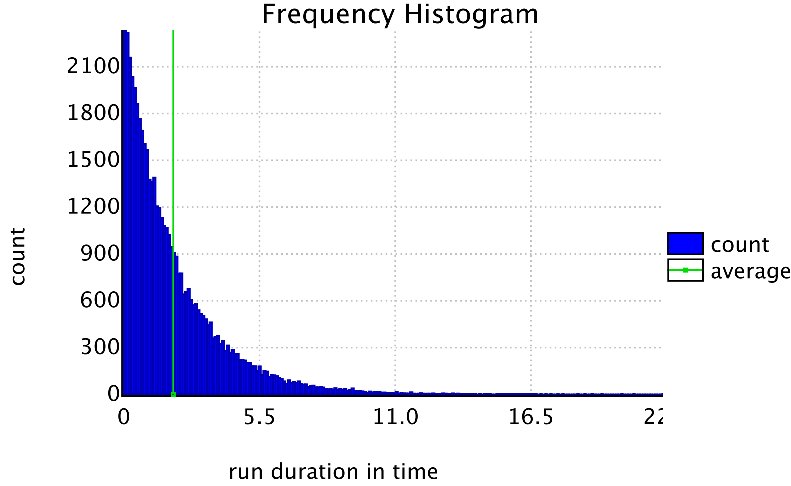


Figure 4.12: Exponential distribution of reachability time.

4.5 Forwarder Module

The *Forwarder* model is changeable in the framework, which is in charge of translating protocols and forwarding messages in the IVN systems. There are many works that propose scheduling algorithms to improve forwarding efficiency of the gateway. In this work, we implement a *ForwardController* automaton with some essential functions, such as monitoring messages, converting protocols, and forwarding messages.

Firstly, the *ForwardController* monitors receiving buffers in the CAN and FlexRay *Interface* of the nodes that are connected to it in real time. When a message is received by a buffer, the *ForwardController* inspects all receiving buffers to forward messages based on a scheduling policy. The *ForwardController* adopts an alternate scheduling for forwarding CAN and FlexRay messages. Fig. 4.13 shows the fragment of *ForwardController* automaton for monitoring *Interface*. When a message is stored in a receiving buffer, the automaton gets a synchronization channel from protocol models while leaves the location `idle`, and goes to the urgent location `monitor_interface`. The following three transitions from this location indicate the scheduling policy of the gateway. When there are no messages in the *Interface*, the automaton returns to the location `idle`. `CRMsgCounter` counts the number of CAN message received, and `FRRMsgCounter` counts the number of FlexRay message received.

- There are only FlexRay messages in the *Interface*. The automaton forwards a FlexRay message with the smallest identifier number to a CAN environment.

- There are only CAN messages in the *Interface*. The automaton forwards a FlexRay

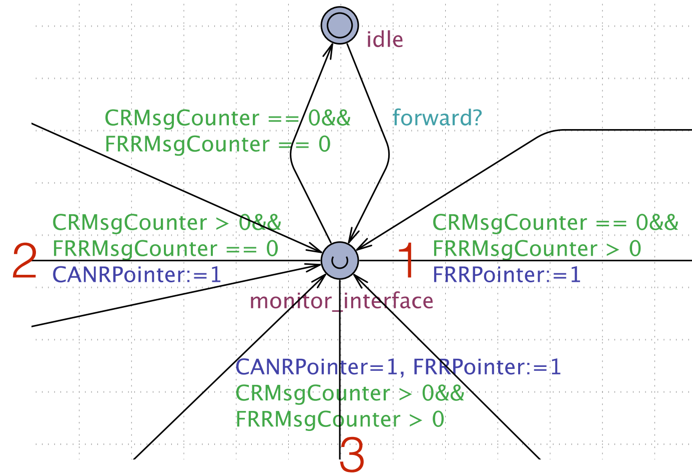


Figure 4.13: GatewayController fragment for monitoring the *Interface*.

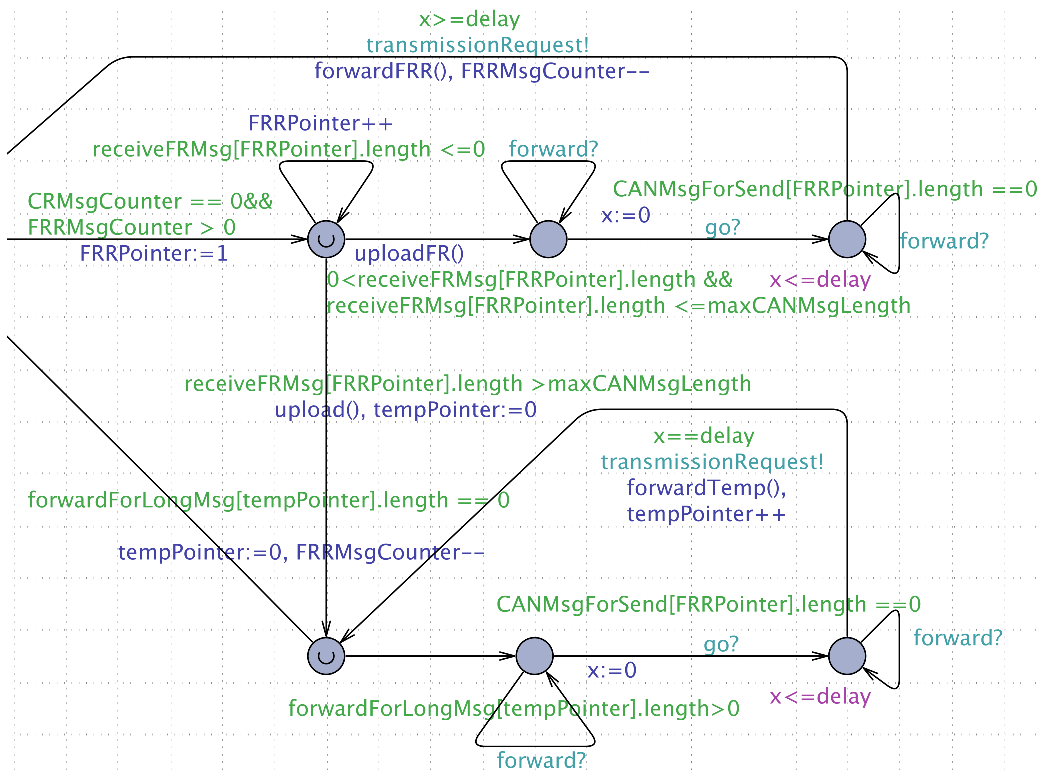


Figure 4.14: *ForwardController* fragment for forwarding messages.

message of the highest priority to a FlexRay environment.

- There are both FlexRay and CAN messages in the *Interface*. The forward controller will deal with a CAN message first and then a FlexRay message.

Taking the first transition as an example, Fig. 4.14 is used to introduce the process of

forwarding messages. When a FlexRay message arrives in its receiving buffer first, the constraint, `CRMsgCounter==0` and `FRRMsgCounter>0` are satisfied. The gateway updates `FRRPointer` to 1, that is the gateway will check receiving buffers of FlexRay *Interface* from the buffer with the smallest identifier. If the first buffer is no message, `FRRPointer` will plus 1 to inspect the next buffer until a buffer has a FlexRay message. If a message was found, the gateway checks whether the message needs to forward to the other environment, by comparing the value of `con` and `dest`. If their value are same, the gateway will looks for next message. Before forwarding the message, the length of the message will be checked by a constraint. If the length meets the configuration of CAN message, this message is directly copied to a temporary buffer by the function `uploadFR()` and waits for forwarding. The gateway checks the sending buffer of CAN *Interface* corresponding to the message. If the sending buffer is empty, the gateway transfers the message to the sending buffer by function `forwardFRR()`, and sends a signal `transmissionRequest` to CAN model. Here we consider a delay for forwarding message in the gateway. After forwarding a message, the gateway goes back to the urgent location `monitor_interface`. If the length of the FlexRay message is beyond the configuration of CAN message, this message will be divided to several messages with the maximum length of CAN message and and same configuration by the function `upload`. These message are stored in temporary buffers, and will be forwarded one by one to the same sending buffer of CAN `Interface` according to the identifier when the sending buffer is empty. As all messages in the temporary buffers are forwarded, the gateway automaton is back to the urgent location `monitor_interface`. A whole *ForwardController* automaton is shown in Fig. 4.15, which combines the monitoring process and forwarding process.

We have covered the construction of each module in the framework. An abstracted system can build its model based on this framework and verify the reachability and response time of messages in UPPAAL. In the next section, we will use an example to show how to build a system model based on the framework.

4.6 An IVN System Design Model

Before constructing an IVN system design model, we need to abstract the original system and know the system design precisely, including the architecture of the IVN system, parameters of the CAN and FlexRay protocols, and the design of the *Environment* and *Forwarder*. Then, each module of the framework can be build in UPPAAL. We will show how to model an IVN system design using the framework by a system design.

The design system has two environments, a FlexRay environment (`FRE1`) and a CAN en-

vironment (CANE2), which are connected by a FlexRay bus, a gateway (GatewayController) and a CAN bus. Fig. 4.16 shows the architecture of the system based on the abstraction. All parts of the system model come from the framework described in section 4.1. The FRE1 has a message msg1 that needs to be sent to the CANE2. The CANE2 also has a message msg2 that needs to be sent to the FRE1. The lengths of msg1 and msg2 are 3 and 2. The communication cycle of FlexRay has two static slots and one dynamic slot, and the slot action point and mini-slot action are same. The msg1 is a static message assigned to the second static slot in the FlexRay, and with an identifier 2 in the CAN. The msg2 is with an identifier 1 in the CAN, and is a dynamic message assigned to the dynamic slot in the FlexRay. The GatewayController performs basic functions to forward messages between FRE1 and CANE2.

According to the description of the IVN system design, the parameter settings in the *Configuration* are listed in Table 4.2, which meet the above description. We build two automata to achieve the requirements of the two environments, FRE1 and CANE2, as shown in Fig. 4.17. They are two sporadic tasks which write msg1 and msg2 following their design. The system design has all types of environments and buses, and messages which are sent to different environments. Thus, this system design is appropriate to illustrate the framework. The design model is set up under the framework in UPPAAL.

4.7 Message Transmission

The framework provides a way to model communication behaviors between CAN subnetwork and FlexRay subnetwork and verify timed properties of messages. Hence, the transmission of messages is also extremely important part of an IVN system model. To describe transmission process of a message within an IVN system model, we summary the message transmission under six topologies. Fig. 4.18 shows six possible system topologies, where E1 and E1 represent a CAN environment or a FlexRay environment, F, F1 and F2 are abstracted gateways, and a CAN bus (CAN) or a FlexRay bus (FlexRay) between forwarders or an environment and a forwarder. For each case, a CAN message or a FlexRay message send from E1 to E2, we use sequence diagrams to show the message transmission, as shown in Fig. 4.19 to Fig. 4.24.

Taking the *Case 1* as an example, we explain the message transmission process in details. There is a FlexRay environment, a forwarder and a CAN environment. We assume that there is a FlexRay message (FRMsg) from the E1 to the E2. The message as a actor requests transmission at some point. Firstly, a FREnv write the FRMsg to a buffer in FRInterface by the function writeFRMsg(). After that, the *FRProtocol* model will

Table 4.2: Configuration of the IVN system design

Name	Value
numberofCANEnv	1
numberofFREnv	1
numberofForwarder	1
numberofCANPro	1
numberofFRPro	1
numberofCANMsg	2
maxLengthofCANMsg	2
numberofFRMsg	2
maxLengthofFRMsg	3
numberofCycle	5
numberofSlot	3
numberofStaticSlot	2
numberofMinislot	10
lengthofStaticSlot	10
lengthofMinislot	2
lengthofNIT	2
slotActionPoint	1
minislotActionPoint	1

transmit the `FRMsg` to FlexRay bus at a slot corresponding the `FRMsgID`. When the slot has finished, the *FRBus* model executes receiving function (`receiveFRMsg()`) and the `FRMsg` is stored to `FRInterface`. Then, the `FRMsg` will be forwarded to the CAN subnetwork by the *Forwarder* and restored to the `CANInterface`. The *CANProtocol* model will transmit the `FRMsgID` according to its priority, and save to the `CANInterface`. Eventually, the `FRMsg` is read by the `CANEnv`.

In *Case 2*, the transmission process from a CAN environment to a FlexRay environment is similar to *Case 1*. *Case 3* and *Case 4* indicate a forwarder centric topology, that is, a forwarder may connect one or more environment using the same bus protocol. In *Case 5* and *Case 6*, there are two forwarders connected in a bus, and the forwarders have other environments connected in other buses. If an abstracted IVN model has a more complex topology, a message needs to go through multiple subnetworks and gateways to reach its destination node, the transmission of the message might repeat this sequence.

Figure 4.15: A *ForwardController* automaton.

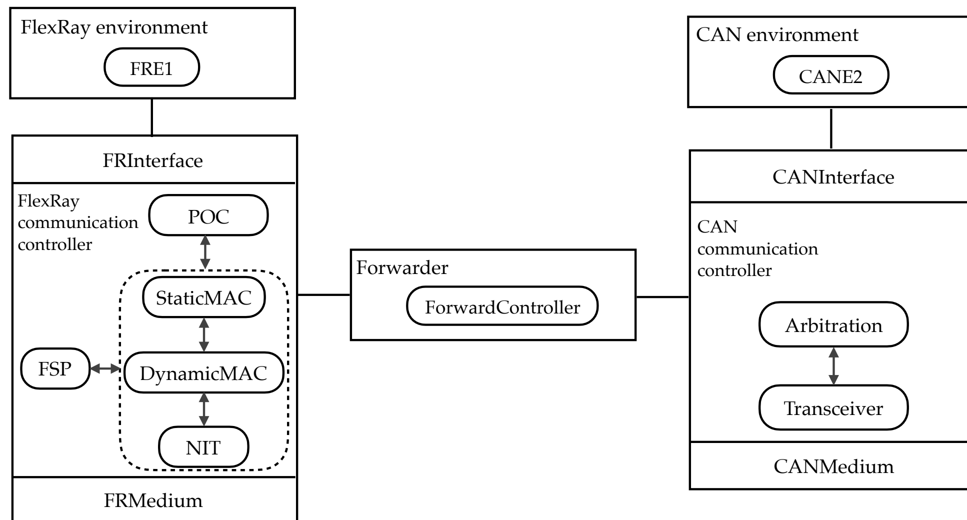
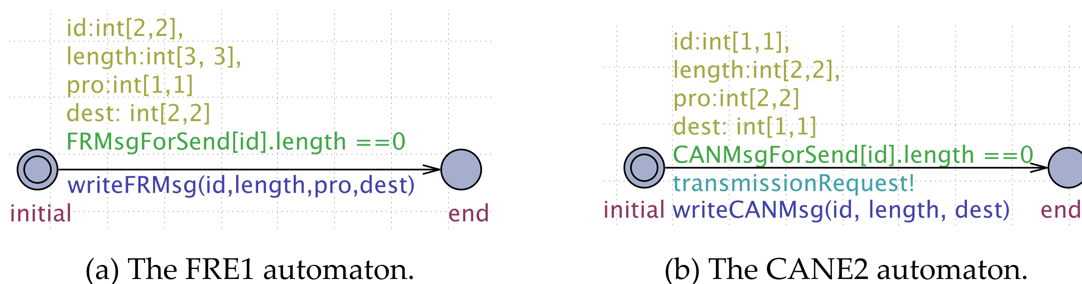


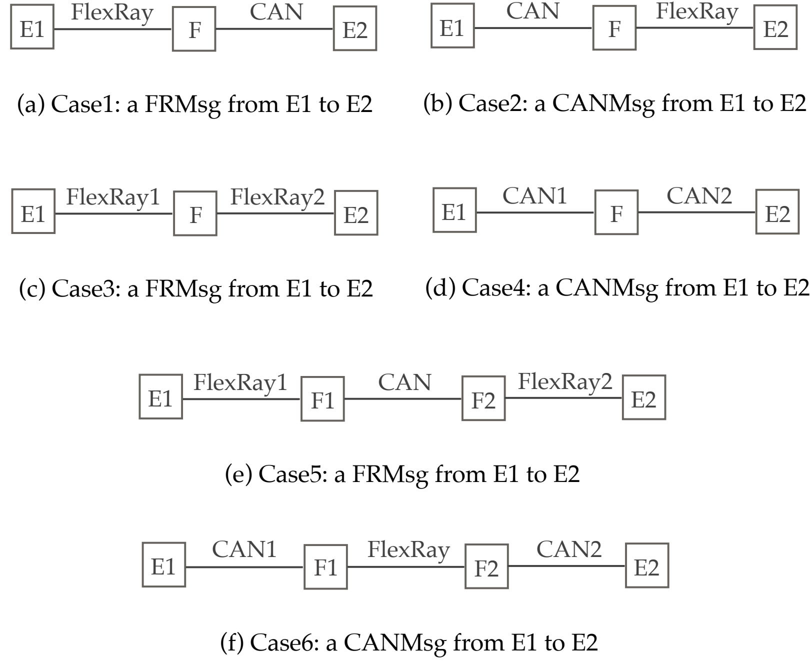
Figure 4.16: Architecture of the IVN system design.



(a) The FRE1 automaton.

(b) The CANE2 automaton.

Figure 4.17: Environment model for the IVN system design.



(a) Case1: a FRMsg from E1 to E2

(b) Case2: a CANMsg from E1 to E2

(c) Case3: a FRMsg from E1 to E2

(d) Case4: a CANMsg from E1 to E2

(e) Case5: a FRMsg from E1 to E2

(f) Case6: a CANMsg from E1 to E2

Figure 4.18: Six possible system topologies.

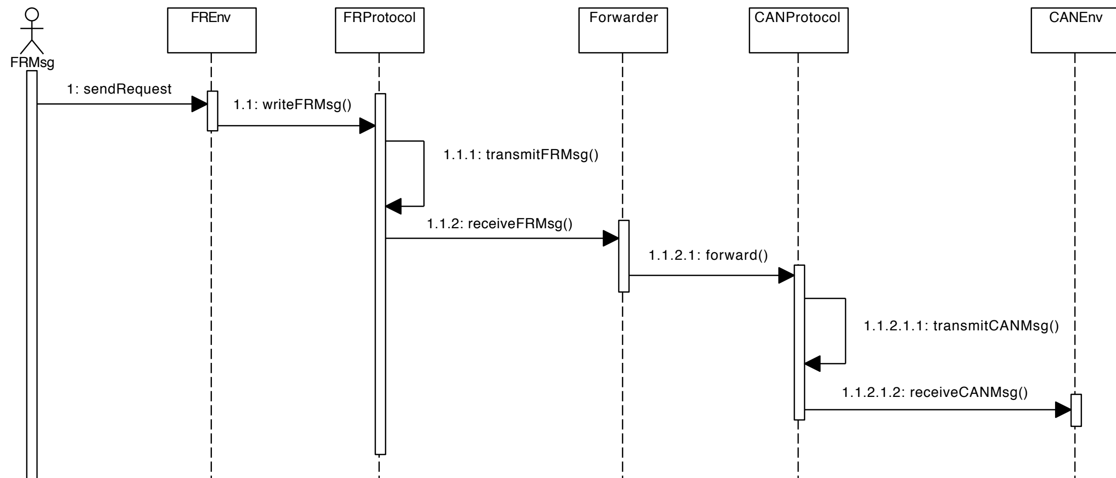


Figure 4.19: The sequence diagram of the Case 1.

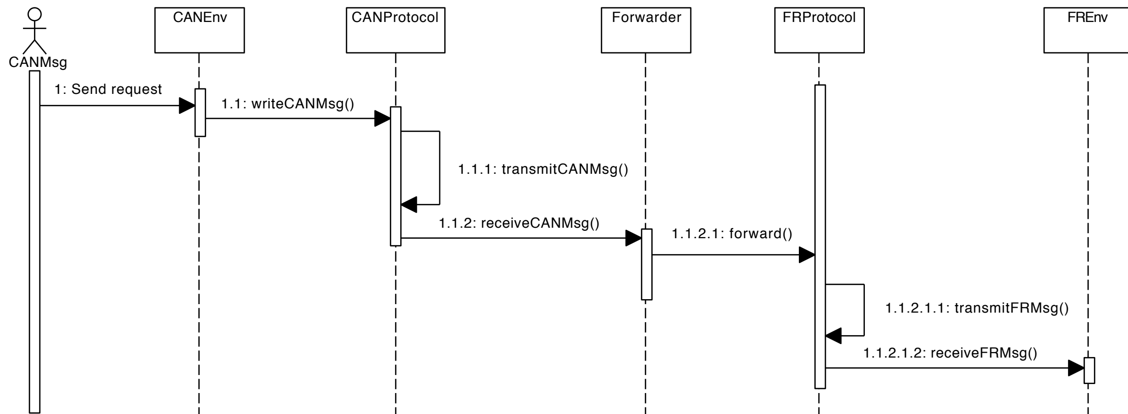


Figure 4.20: The sequence diagram of the Case 2.

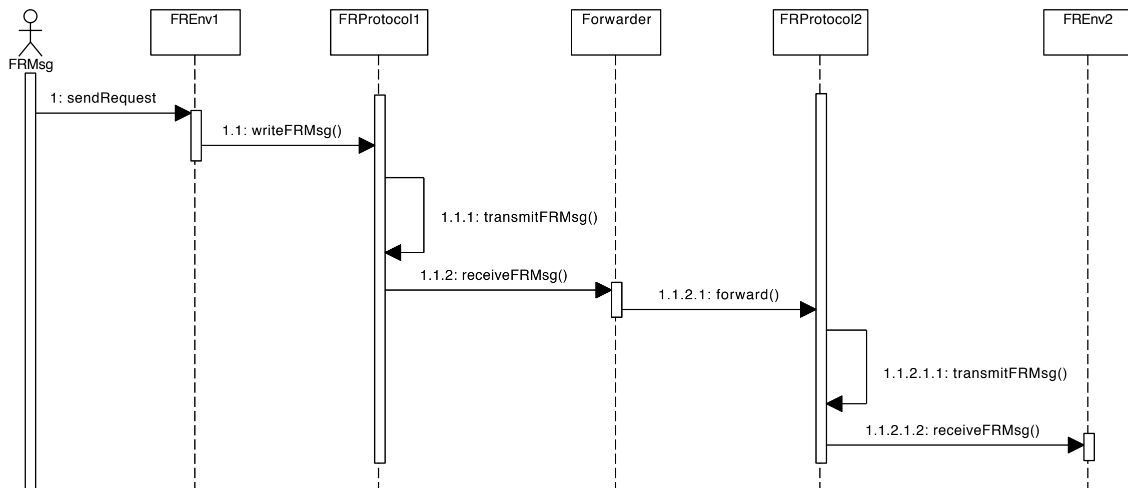


Figure 4.21: The sequence diagram of the Case 3.

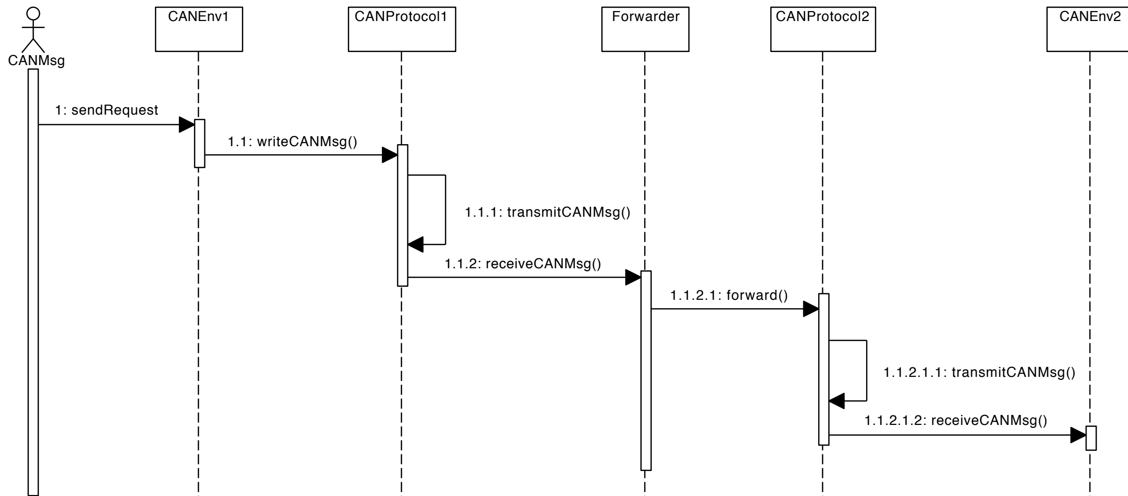


Figure 4.22: The sequence diagram of the Case 4.

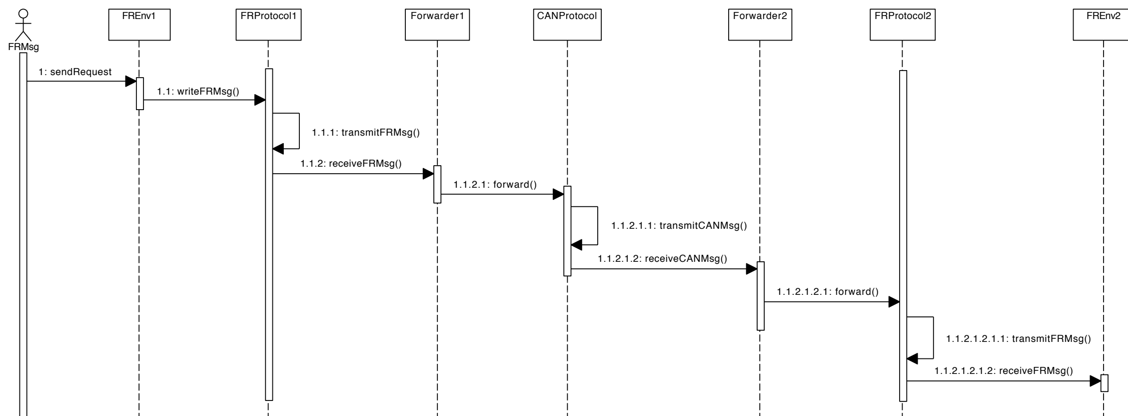


Figure 4.23: The sequence diagram of the Case 5.

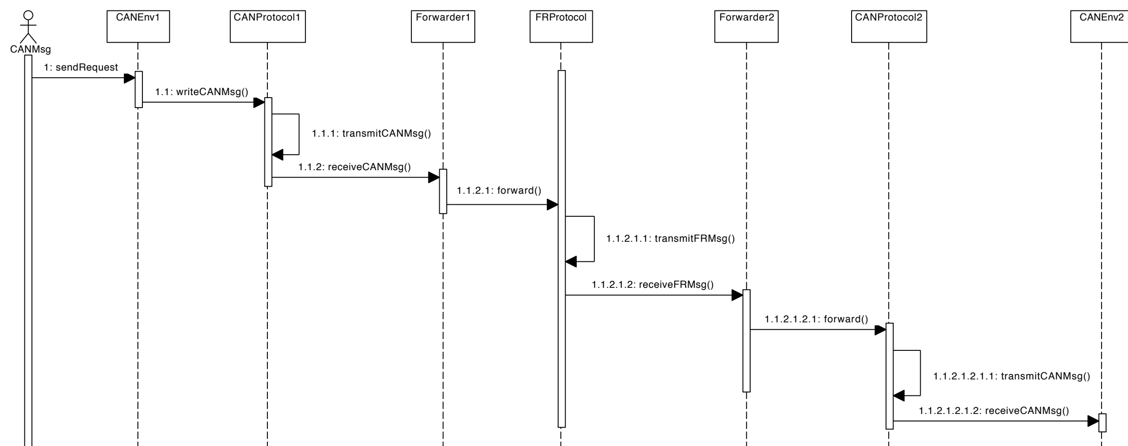


Figure 4.24: The sequence diagram of the Case 6.

Chapter 5

Verification and Evaluation

Based on the framework, we implement some IVN system model to verify properties from two aspects, qualitative verification and quantitative verification. The qualitative verification is complete and exhaustive to guarantees that the framework satisfies a certain property absolutely. We check qualitative representation to verify the validity of the framework and show the reachability and response time of message in an IVN design model by traditional model checking. To show that the framework is reusable, three typical topologies of IVN systems are accomplished and their response times are compared. The quantitative verification is accurate to tackle ubiquitous probabilistic behaviors in the IVN systems. We construct probabilistic IVN design models and check the reachability and response time of messages by statistical model checking. The influence of probabilistic behaviors on system properties is analyzed from quantitative verification results.

5.1 Qualitative Verification

The proposed framework is evaluated in terms of four aspects: 1) a basic but non-trivial design of IVN system is applied to evaluate the validity of the framework by checking a train of essential properties with UPPAAL; 2) we check the reachability and response time of messages in the IVN system; 3) the reusability of the framework is illustrated by designing three typical IVN system topologies and comparing their timed properties; 4) the performance of the framework is tested by a design model.

5.1.1 Validity of the IVN System Model

In the framework, the protocol module is vitally critical for transmitting message. Hence, we firstly check some properties to guarantee the correctness of the CAN and FlexRay

model using the system design model.

- **Check1:** Is the system model deadlock free?

  ```
  Query1: A[] not deadlock
  ```

  This query is satisfied. The system model will never be deadlock. The deadlock freeness is an important property for verifying the credibility of the model.

- **Check2:** Are the messages transmitted and received in the assigned slots in FlexRay?

  ```
  Query2: A[] forall(i:int[1,3])(FRMsgForReceive[i].length > 0) imply
          (commStatus.slotCounter==i)
  ```

  This query is satisfied. In this model, we set up three slots in FlexRay communication cycle. All messages are correctly received in the allocated slots. The `msg1` will be received in slot 2, and the `msg2` will be received in slot 3. Thus, the FlexRay model can control the message sending and receiving processes following the slot number.

- **Check3:** Each message transmission monopolize the bus. Does only one buffer receive the transmitted message in a slot?

  ```
  Query3: A[] forall(i:int[1,3]) forall (j:int[1,3])
          (FRMsgForReceive[i].length>0)&&(FRMsgForReceive[j].length>0)
           imply (j=i)
  ```

  ```
  Query4: A[] forall(i:int[1,2]) forall (j:int[1,2])
          (CANMsgForReceive[i].length>0)&&(CANMsgForReceive[j].length>0)
           imply (j=i)
  ```

  The two queries are satisfied. It means that only one message will be received by a buffer of FlexRay or CAN. If two receiving buffers in *Interface* get messages at the same time, the identifier of the messages should be same. That is only one message utilizes bus in a moment of the system.

- **Check4:** Does the application model output messages successfully?

  ```
  Query5: E<> forall(i:int [1,3])(FRMsgForSend[i].length>0)
  ```

  ```
  Query6: E<> forall(i:int [1,2])(CANMsgForSend[i].length>0)
  ```

  The two queries are satisfied. The application model can write messages to *Interface*.

- **Check5:** Will the message in the sending buffers be sent?

```
Query7: forall(i:int [1,3])((FRMsgForSend[i].length>0) imply
        (FRMsgForSend[i].length==0)
```

```
Query8: forall(i:int [1,2])((CANMsgForSend[i].length>0) imply
        (CANMsgForSend[i].length==0)
```

The two queries are satisfied. The CAN and FlexRay models can finish transmissions for all messages.

We examined the `Check1-5` to ensure whether messages were stored in the interface correctly, that is, messages waiting in the sending buffers are transmitted by the CAN/FlexRay model and the transfer process conforms to the protocol specifications. The `Check1` confirms that the IVN system design model is no deadlock, and can perform message transmissions. The `Check2` states that, while a receiving buffer contains a message, the identifier of the message should equal to the current slot number, which means message transmission occur in a correct slot. The `Check3` states that there is only one message being sent in any time. The `Check4` states that the application models can successfully write messages to the sending buffers of CAN and FlexRay. The `Check5` means that all of messages can be transmitted. These queries state that FlexRay model accords with TDMA communication scheme, and the FlexRay model and CAN model satisfy protocol specifications regarding normal message transmissions.

Next, the message transmission process was checked, since we have to ensure that these nodes connection and communication correctly. The transmission processes of the `msg1` and `msg2` in the CAN and FlexRay environments are monitored by following queries:

- **Check6:** Can the `msg1` be transmitted from *E1* to *E2*?

```
Query9: E<> (FRMsgForSend[2].length > 0)
```

```
Query10: (FRMsgForSend[2].length > 0)-->(FRMsgForSend[2].length == 0)
```

```
Query11: (FRMsgForSend[2].length>0)-->(busFlexRay.id == 2 &&
         busFlexRay.length > 0)
```

```
Query12: (busFlexRay.id == 2&&busFR.length == 0)-->
         (FRMsgForReceive[2].length > 0)
```

```
Query13: (FRMsgForReceive[2].length > 0)-->
         (CANMsgForSend[2].length > 0)
```

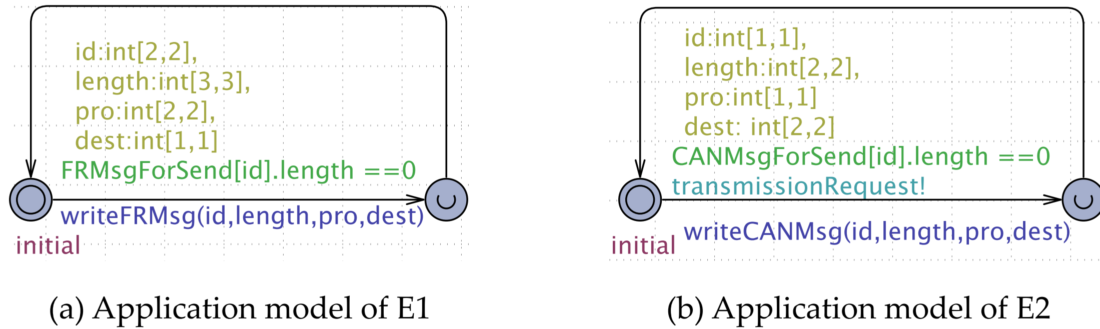
```
Query14: ((CANMsgForSend[2].length > 0)-->
         (CANMsgForSend[2].length == 0)
```

```
Query15: (CANMsgForSend[2].length > 0)-->
         (busCAN.id == 2&&busCAN.length > 0)

Query16: (busCAN.id == 2&&busCAN.length == 0)-->
         (CANMsgForReceive[2].length > 0)
```

- **Check7:** Can the msg2 be transmitted from *E2* to *E1*?

```
Query17: E<> (CANMsgForSend[1].length > 0)

Query18: (CANMsgForSend[1].length > 0)-->
         (CANMsgForSend[1].length == 0)

Query19: (CANMsgForSend[1].length>0)-->
         (busCAN.id == 1&&busCAN.length > 0)

Query20: (busCAN.id == 1&&busCAN.length == 0)-->
         (CANMsgForReceive[1].length > 0)

Query21: (CANMsgForReceive[1].length > 0)-->
         (FRMsgForSend[3].length > 0)

Query22: (FRMsgForSend[3].length > 0)-->(FRMsgForSend[3].length == 0)

Query23: FRMsgForSend[3].length>0)-->(busFlexRay.id == 3 &&
         busFlexRay.length > 0)

Query24: (busFlexRay.id == 3&&busFR.length == 0)-->
         (FRMsgForReceive[3].length > 0)
```

Query 9-16 check the basic functionalities of the framework. The msg1 is written by the FlexRay task model until it is received by *E2*. Query9 indicates the message is written to a FlexRay sending buffer in the *Interface*. Query10 and Query11 say the message in the sending buffer is sent to FlexRay bus. Query12 says the message is received by the corresponding receiving buffers in FlexRay interface. Query13 means that the gateway forward the message from FlexRay to CAN. Afterwards, the message is transferred to corresponding sending buffers of the other environment. That is the gateway can forward message between the FlexRay and CAN environments. Query14 and Query15 indicate the message can be transmitted to CAN bus by CAN model. Query16 says the message is received by *E2*. Similarly, we also check the msg2 that is transmitted from *E2* to *E1* using Query17–Query24. These queries are satisfied in the IVN system design model. The Check6 and Check7 verify the process of messages from FlexRay to CAN and from CAN to FlexRay respectively. We can conclude that the proposed framework is valid for normal message transmissions under the CAN and FlexRay protocols.



(a) Application model of E1

(b) Application model of E2

Figure 5.1: Observer for checking response time.

5.1.2 Reachability

To verifying safety property of the IVN system, the reachability of messages is considered. When there are many messages transmitted between a CAN environment and a FlexRay environment, it is essential to verify whether all messages can be received or if there is a message lost. Here, we also use the IVN design model but change the application model. For the system with two messages that need to be sent repeatedly, we used two cyclic tasks to send `msg1` and `msg2` in the system design model. The results of `Query29` and `Query30` show that `msg1` and `msg2` cannot always be received, which are not satisfied. As the priority of `msg1` is lower than that of `msg2`, it cannot be sent in the CAN environment. Because messages are frequently sending through the gateway, the gateway becomes congested. So some messages are lost during transmission.

- **Check10:** Can all messages be received by their destination?

  `Query29: (observerMsg1.sent --> observerMsg1.received)`

  `Query30: (observerMsg2.sent --> observerMsg2.received)`

5.1.3 Response Time

A main feature of the framework is the description of behaviors with timed constraints. The transmission time of messages can be checked using an observer model that monitors specific model states. A sender task can release messages repeatedly, and a receiver task receives messages immediately when the interface detects the presence of data.

From a message waiting in a sending buffer to the message received from a receiving buffer by the destination node, this process includes the transmission time on buses as well as the waiting time in the *interface*, called response time. We can check a train of

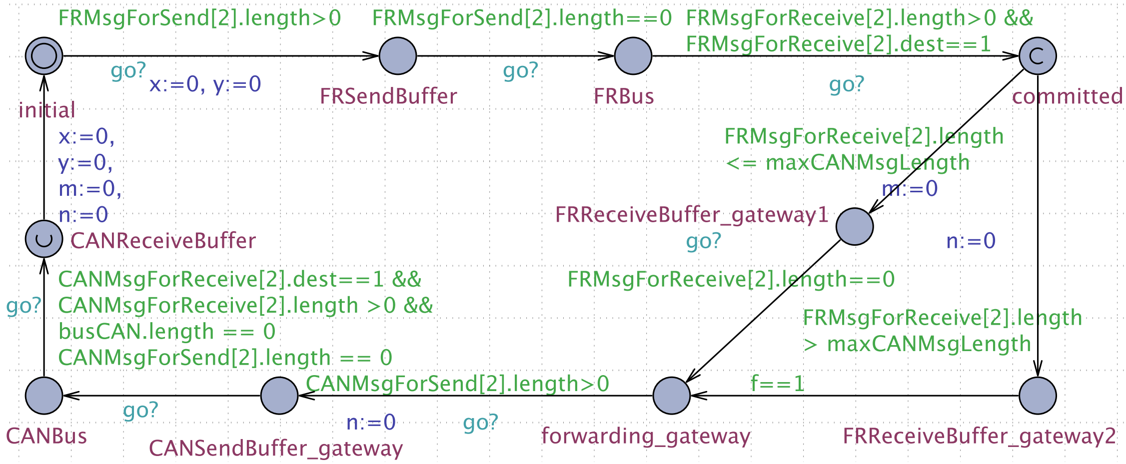


Figure 5.2: Observer for checking response time.

timed properties to obtain the best-case response time (BCRT) and worst-case response time (WCRT) of messages.

In the system design model, `msg1` is sent from the FlexRay environment *E1* and transmitted to the CAN environment *E2*. We check the response time of `msg1` with an observer model as shown in Fig. 5.2.

The observer model monitors the `msg1`. `Query25`–`Query28` verify BCRT and WCRT for `msg1`.

- **Check8:** What is the best response time of the `msg1`?

  `Query25: E<>(observerMsg1.CANReceiveBuffer && observerMsg1.x<34)`

  `Query26: A[](observerMsg1.CANReceiveBuffer imply observerMsg1.x>=34)`

- **Check9:** What is the worst response time of the `msg1`?

  `Query27: E<>(observerMsg1.CANReceiveBuffer && observerMsg1.x>94)`

  `Query28: A[](observerMsg1.CANReceiveBuffer imply observerMsg1.x<=94)`

In `Check8`, `Query25` and `Query26` indicate the best case response time of `msg1`. The `Query25` means if there is a response time less than `34` time units. The `Query26` means if all response time of `msg1` are greater than or equal to `34` time units. The checking results are the `Query25` is not satisfied, and the other one is satisfied. So `34` is the BCRT of `msg1`. In `Check9`, we obtain the worst case response time of `msg1`. The `Query27` is not satisfied, that is no response time of `msg1` is greater than `94` time units. The `Query28` is satisfied, that is all response time of `msg1` is less than or equal to `94` time units. As a result, the WCRT of `msg1` is `94`. The values of BCRT and WCRT in a query are determined by trial

and error based on the parameters of the IVN system currently, such as message length, length of cycle of the task and communication, and delay time in gateway.

There is a special case, when the length of `msg1` is greater than the maximum length of CAN message. We considered `msg1` to have a length of `9`, which is sent from a FlexRay environment to a CAN environment. In this case, the message is divided into a message of length 8 and a message of length 1. The BCRT and WCRT is `100` and `160`. In the same way, we can check the BCRT and WCRT of a message using these four queries. Thus, the framework can check the timed properties with the help of the observer model.

5.2 Quantitative Verification

In the last section, we checked validity, reachability and response time of IVN systems using classical model checking. However, the performance is limited by complexity. The system model can only handles with several messages with different identifiers. A practical IVN system has a large number of messages transmitted, and these messages are sent by stochastic tasks. The model checking method is not able to perform stochastic behaviors and checks properties along with state space explosion problem for complicated systems. Consequently, in this section, we present application models with probability based on the CAN model of the framework and verify probabilistic properties to improve the previous results.

5.2.1 Application Model with Probability

In the IVN systems, FlexRay protocol defines a hard real-time communication with a fixed length communication cycle, and messages are sent in their special time slots. CAN protocol is a soft real-time communication with a fixed priority scheduling, and timing of message transmission is undecidable. CAN subsystem will affect response time of the entire IVN system. Hence, we first consider CAN system as a checking object to do quantitative verification in this work.

The probabilistic application models are mentioned in section 4.4.2. For constructing a CAN system model, we also need CAN protocol model and CAN interface as mentioned in chapter 3. The stochastic model of CAN system can be built by the probabilistic task automata, Interface and CAN protocol model. Here we make a comparison of models between model checking and statistical model checking, as shown in Table 5.1. According to the structure of CAN nodes, each layer has be compared.

- Application model are described by task automata. When we checked the response

Table 5.1: Comparison of models between model checking and statistical model checking

Model	Model checking	Statistical model checking
Application	1) Tasks send messages once. 2) The triggering time is satisfying a constraint.	1) Tasks send messages repeatedly. 2) The triggering time is based on a probability distribution.
Interface	Same	Same
Protocol	1) Arbitration automaton is same. 2) Only a transceiver automaton deals with all messages.	1) Arbitration automaton is same. 2) Messages with the same identifier have a transceiver automaton.

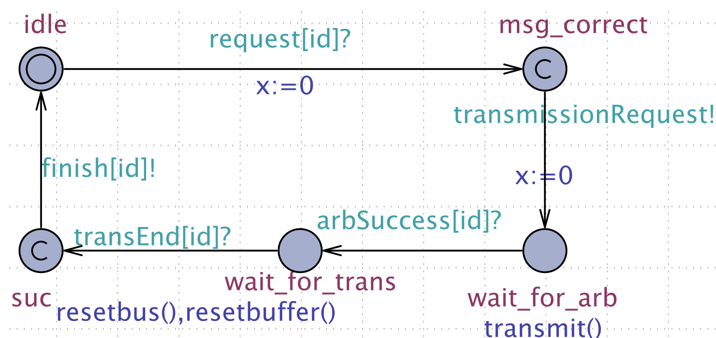


Figure 5.3: Transceiver automaton for statistical model checking.

time of messages, task models send each message once, and the timing of writing messages is satisfying a constraint (`CANMsgForSend[id].length == 0`).

- Interface model are same in both model checking and statistical model checking.
- For protocol model, the arbitration automaton is no change. The transceiver automaton will be installed to each message identifier as shown in Fig. 4.11, since the quantitative verification has to clearly know states of each message. The transceiver monitors messages from writing to receiving.

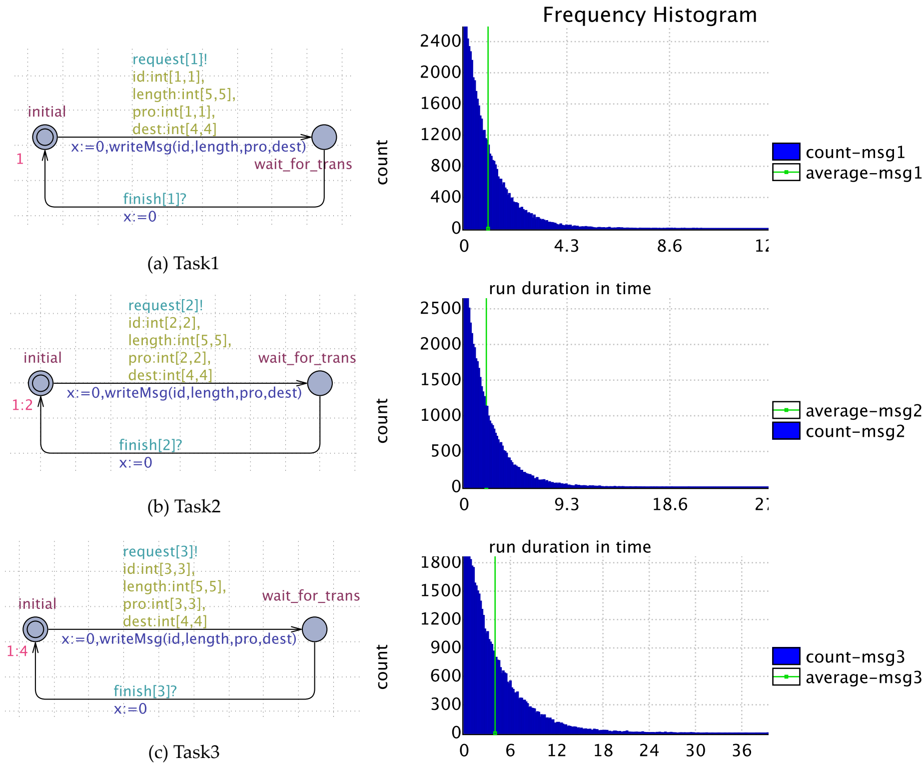


Figure 5.4: Tasks automata and distributions of reachability time in Design 1.

5.2.2 Reachability and Response Time

In the qualitative verification, we have checked the reachability of messages using two cyclic tasks, but the results of `Query29` and `Query30` are not satisfied. There are messages were not received. Also, the response time (BCRT and WCRT) of message can be obtained by a series of queries. However, we cannot get more information to analyze the reachability and response time from the qualitative verification. Using the quantitative verification, the system can be simulated for finitely many runs, and a statistical result is given if the property is satisfied with some degree of confidence. Consequently, we implement two CAN system design models with different probabilities to verify and analyze the reachability and response time.

- **Design 1:** This CAN system has three nodes, and each node has a task to write messages with delays according to an exponential distribution. Fig. 5.4 shows three task automata and exponential distribution of their arrival to `wait_for_trans`. The rate of exponential distribution is `1`, `1:2` and `1:4` for each task, and the expected value of the exponential density function is `1`, `2` and `4`. That is, the frequency of task1, task2 and task3 is `1`, `2` and `4` for delivering messages.

– **Reachability:** We check a probability that is a message reaches a state where it has finished its transmission before 100 time units during the run of the system. The state indicates the message has been received. Three queries for each message are verified in SMC-UPPAAL as following:

```
Query31: Pr[<=100](<>transceiver(1).suc)                [0.9998,1]
Query32: Pr[<=100](<>transceiver(2).suc)                [0.9998,1]
Query33: Pr[<=100](<>transceiver(3).suc)     [0.617107, 0.627107]
```

The `Query31` and `Query32` are satisfied with a probability close to 100% , that means the `msg1` and `msg2` can be received almost. The `msg3` with lower priority, will wait for arbitration a long time, during the 100 time units `msg3` is received with lower probability.

– **Response time:** Although the messages can be received within 100 time units, we have to know what the response time of the messages is and how many present messages can be received within 10 time units, for a real-time system. The following three queries check response times of the messages.

```
Query34: Pr[<=100](<>transceiver(1).suc && transceiver(1).x <=10)
                                                         [0.9998,1]
Query35: Pr[<=100](<>transceiver(2).suc && transceiver(2).x <=10)
                                                         [0.9998,1]
Query36: Pr[<=100](<>transceiver(3).suc && transceiver(3).x <=10)
                                                [0.190766,0.200766]
```

According to the results, the response time of almost `msg1` and `msg2` is less than or equal 10 time units. Most of `msg3` cannot received within 10 time units. Then, we can check the probability density of the response time and the average response time of the response time by Fig. 5.5. Based on the cumulative probability distribution, `msg1` and `msg2` are received during a short time interval.

- **Design 2:** This CAN system has three nodes, and each node has a task to write messages with delays according to an uniform distribution. Fig. 5.6 shows three task automata and uniform distribution of their arrival to `wait_for_trans`. The

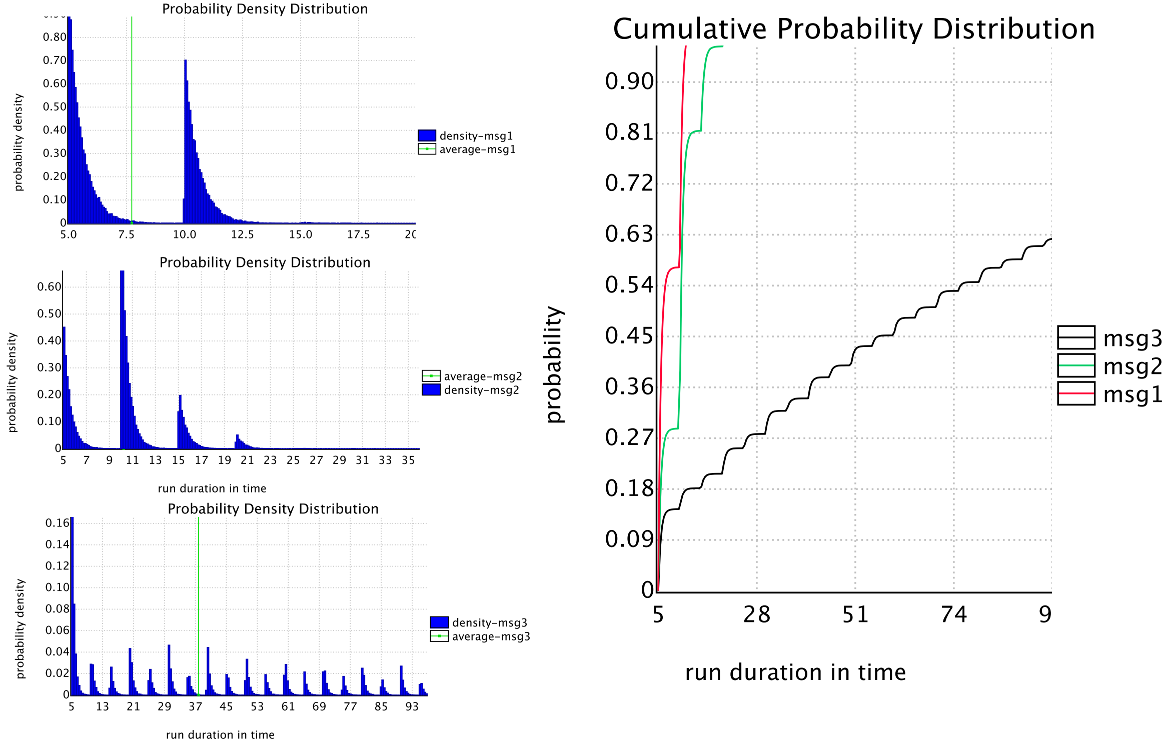


Figure 5.5: Probability density distribution of response time in Design 1.

expected value of the uniform density function is 1, 2 and 4, that is, the frequency of task1, task2 and task3 is 3, 5 and 7 for delivering messages.

– **Reachability:** Similarly, we check probability of message reachability. There are three queries for each message are verified in SMC-UPPAAL as following:

```
Query37: Pr[<=100](<>transceiver(1).suc)                 [0.9998,1]
Query38: Pr[<=100](<>transceiver(2).suc)                 [0.9998,1]
Query39: Pr[<=100](<>transceiver(3).suc)        [0.995883,0.997883]
```

The `Query37` and `Query38` are satisfied with high probability, that means the `msg1` and `msg2` can be received mostly. Although the `msg3` has the lowest priority, the probabilistic reachability is better than Design 1.

– **Response time:** In the same way, the response time has been checked by following queries.

```
Query40: Pr[<=100](<>transceiver(1).suc && transceiver(1).x <=10)
                                                          [0.9998,1]
```

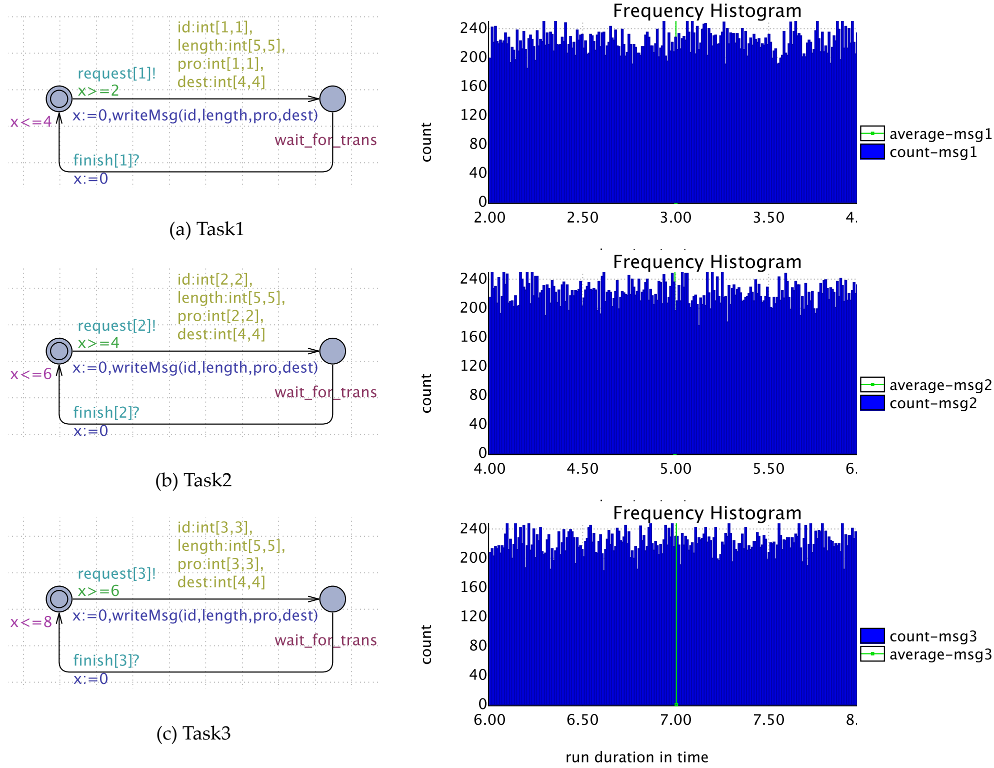


Figure 5.6: Tasks automata and distributions of reachability time in Design 2.

```
Query41: Pr[<=100](<>transceiver(2).suc && transceiver(2).x <=10)
                                                        [0.9998,1]
Query42: Pr[<=100](<>transceiver(3).suc && transceiver(3).x <=10)
                                                   [0,0.00199982]
```

According to the results, the response time of almost `msg1` and `msg2` is less than or equal 10 time units. Few `msg3` can received within 10 time units. Then, we can check the probability density of the response time and the average response time of the response time by Fig. 5.7. In the cumulative probability distribution figure, `msg1` and `msg2` are received during a short time interval as Design 1, but `msg3` has a higher probability to transmit than Design 1.

5.3 Evaluation

In this section, we will evaluate the framework from multiple aspects. Firstly, we will discuss whether the framework conforms to protocol specification, and checked properties are credible. Secondly, we estimate the applicability and reusability of the framework. At

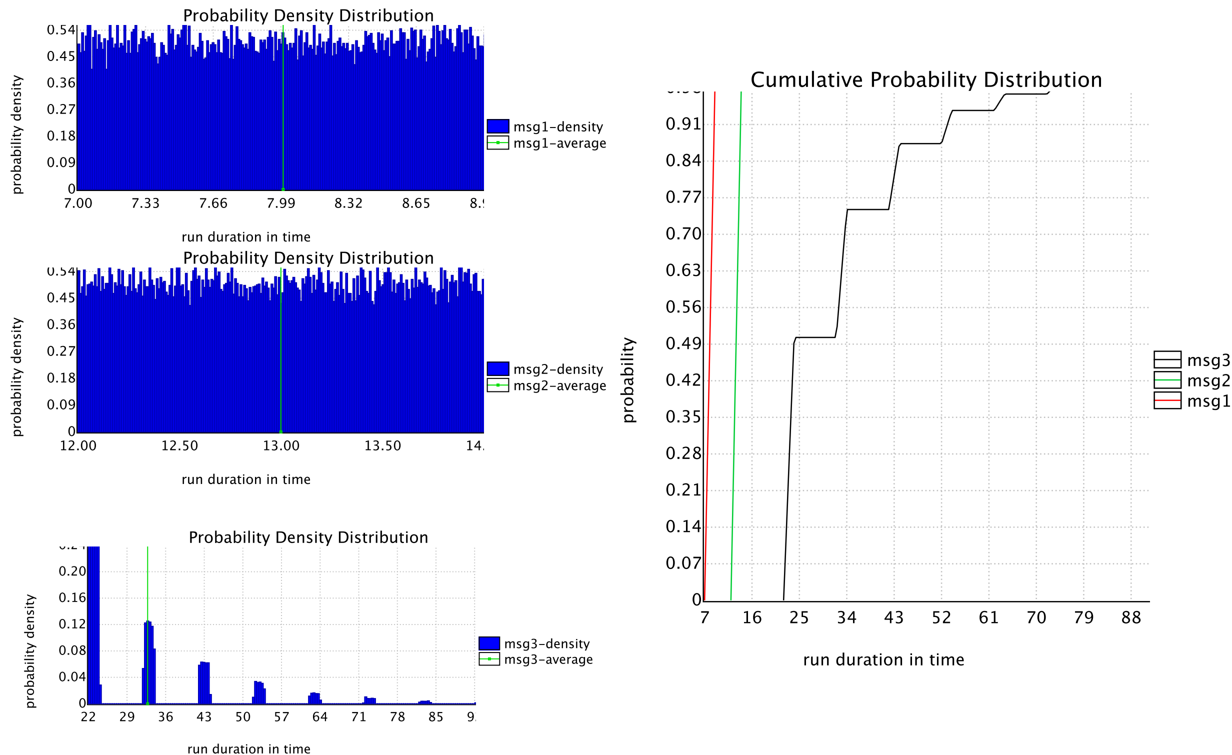


Figure 5.7: Probability density distribution of response time in Design 2.

last, we show the performance in different verifications.

5.3.1 Validity of the Abstraction

In order to discuss the preservation of the framework, we illustrated it from two aspects: abstracting the protocol model from protocol specifications and the architecture of IVN systems.

We abstracted the description of the CAN protocol and FlexRay protocol according to their specifications. It is extremely important that the *CAN protocol* model and *FlexRay protocol* model perform communication behaviors directly. We checked properties from protocol specifications that are listed in section 5.1.1. These properties are kept in our system model, but there are some properties that cannot be verified because of the abstraction the specifications. We list as many properties as possible in Table 5.2 and Table 5.3 from CAN specification and FlexRay specification to evaluate the framework. In Table 5.2, No.1 to No.11 can be satisfied in our framework, but No.12 to No.13 can not be checked. No.12 belongs to deal with error message, No.13 relates to remote frames that are not considered in this work, and No.13 is a set of properties related to physical

Table 5.2: Properties in CAN specifications

No.	Property	Result
1	If and only if a node may send a message in a certain time.	Y
2	Messages sent by environments can participate in the arbitration, if and only if the bus is free.	Y
3	Multiple nodes can participate in the arbitration at the same time.	Y
4	After the arbitration begins, there must be a message that wins the arbitration.	Y
5	If a node is sending a message, the bus state must be busy.	Y
6	A message that wins the arbitration has the highest priority.	Y
7	While a transmission ends, the bus state becomes to idle.	Y
8	Each node in a same cluster can receive same messages.	Y
9	Every node has to recognize messages before dealing with messages.	Y
10	Every node can only receive messages that are consistent with the filter setting.	Y
11	The clock of every node is synchronous.	Y
12	If a corrupted frame is detected by a node, the message destroyed and retransmitted.	-
13	If a node send a remote frame, the node will eventually receive the message requested.	-
14	Error signals can be detected.	-

layer. Similarly, Table 5.3 also lists properties that can be verified, and some types that can not be checked.

The CAN and FlexRay models in UPPAAL are established based on the CAN and FlexRay specifications. We have mainly extracted the content related to the communication behaviors in the case that the transmission is correct, as we listed the preserved properties in the tables. However, those signal-level behaviors, as well as those behaviors related to error handling, are not considered. Therefore, these related properties cannot be preserved in our model.

On the other hand, we checked reachability and response time of messages in IVN system models. These results are based on the abstraction. If we had not simplify the architecture of IVN systems, the checking results would have been different. We try to build a group of IVN systems without the architecture abstraction, and check

Table 5.3: Properties in FlexRay specifications

No.	Property	Result
1	Messages sent by nodes according to slot number.	Y
2	If and only if a node may send a message in a certain slot.	Y
3	All messages transmit in their fixed slot.	Y
4	Messages can only be transmitted after an action point in each slot.	Y
5	If there is no minislots in the communication cycle, the cycle has not a dynamic segment.	Y
6	If there are no more slots in a cycle, no more messages will be transmitted during this cycle.	Y
7	When the last slot ends and there are still minislots, these minislots have to be consumed to start the next cycle.	Y
8	As soon as a message is transmitted, the bus state is marked as TRANS.	Y
9	When a transfer is finished, the bus state is changed to CHIRP.	Y
10	If and only if a slot ends, the bus state becomes IDLE.	Y
11	In the dynamic segment, if there is no messages need to be sent, the slot only takes one minislot.	Y
12	If a node is sending a message, the bus state must be busy.	Y
13	A message transmits according to the time slot that it defines.	Y
14	While a transmission ends, the bus state becomes to idle.	Y
15	Each node in a same cluster can receive same messages.	Y
16	Every node has to recognize messages before dealing with messages.	Y
17	Every node can only receive messages that are consistent with the filter setting.	Y
18	The clock of every node is synchronous.	Y
19	If a corrupted frame is detected by a node, the message destroyed and retransmitted.	-
20	If a node send a remote frame, the node will eventually receive the message requested.	-
21	Error signals can be detected, and it identified as an error frame.	-

reachability and response time of messages in each systems. Now we restore a subnetwork of an environment with different topologies, as shown in Fig. 5.8. Fig. 5.8 (a) shows an abstracted IVN system that describe in section 4.6, and the *E2* is an abstracted CAN subsystem. We replace the subsystem *E2* with different topologies. In Fig. 5.8 (b), the subsystem is with bus topology, and *N1*, *N2* and *N3* are nodes connected in a CAN bus. In Fig. 5.8 (c), the subsystem is with star topology, and *N2* and *N3* connect to *N3*, and *N3* is connected to the forwarder *F*. In Fig. 5.8 (d), the subsystem is with point to point topology, and *N1*, *N2* and *N3* are nodes connected by three CAN buses.

In these system, the *F* directly connects to the *N1* of the subnetwork, so we call it a connection node. Assuming that there is a FlexRay message (`FRMsg`) sending from *E1* to *N3*, we verify response time of the `FRMsg` from *E1* to the connection node (*N1*) and the destination *N3* in every architecture. Table 5.8 shows results compared with the abstracted systems.

- When the subnetwork using a bus topology, the response time of the `FRMsg` is same as the abstracted system. The BCRT and WCRT of the message to reach the *N1* and *N3* are uniform. All nodes can receive the message at the same time, since these nodes are connected to the *F* in a CAN bus.

- When the subnetwork using a star topology, the response time of the `FRMsg` to the *N3* is longer than the abstracted system, because the *N1* relays the message again to the destination node (*N3*). The BCRT and WCRT, it takes for the message to reach the *N1*, is the same as the abstract system.

- When the subnetwork using a point to point topology, the response time of the `FRMsg` is more long, because the *N1* and *N2* relay the message twice to the destination node *N3*. But the BCRT and WCRT for reaching the *N1* is not changed.

These results suggest that the abstraction of the architecture of the systems can preserve the timed property until the first node directly connected to the gateway, as subnetworks are abstracted as environments connected to the original gateway. But the inside of the subnetwork can not be preserved, we can use single protocol to verify it.

5.3.2 Applicability

In section 2.5.3, we have discussed topologies of IVN systems, and mention three types based gateway, central, backbone and daisy chain topologies. The response time in these topologies will be affected by the number of environments and the number of message

Table 5.4: Comparison of response times of the `FRMsg` in different topologies.

Subnetwork	From *E1* to *N1*		From *E1* to *N3*	
	BCRT	WCRT	BCRT	WCRT
Star topology	34	74	64	104
Bus topology	34	74	34	74
Point to point topology	34	74	94	134

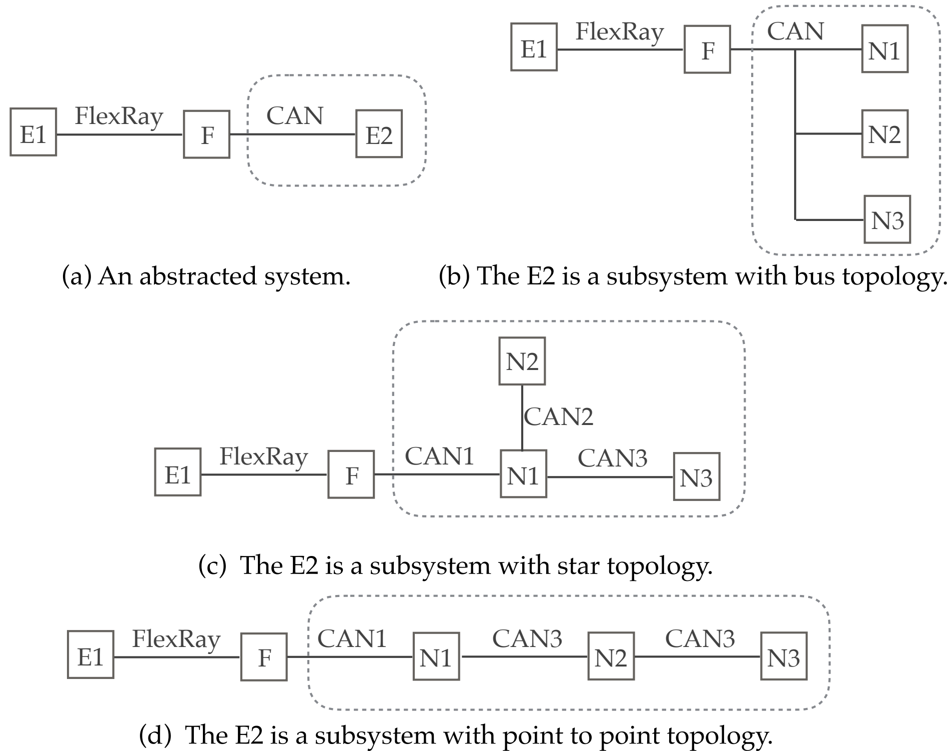
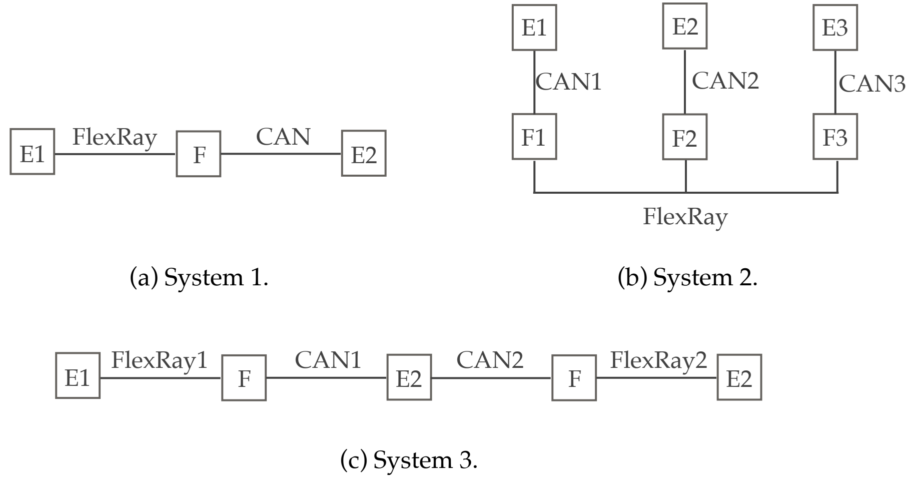


Figure 5.8: Restored architectures of a subsystem.

identifiers. In order to estimate the applicability of the framework, for each of the three IVN system topologies, we implemented three cases and compared their response time (see Table 5.5. Note that CAN, FlexRay, and the forwarders and interfaces had the same parameter settings.

- In *Case 1*, two environments make up the three topologies. The topologies of *Case 1* are shown in Fig. 5.9, and the application model of each topology is shown in Fig. 5.10.

  – The central topology and daisy chain topology have same structure when the system only has two environments. `msg1` (`id=2`, `length=3`) is transferred from



(a) System 1.

(b) System 2.

(c) System 3.

Figure 5.9: Topologies in *Case1*.

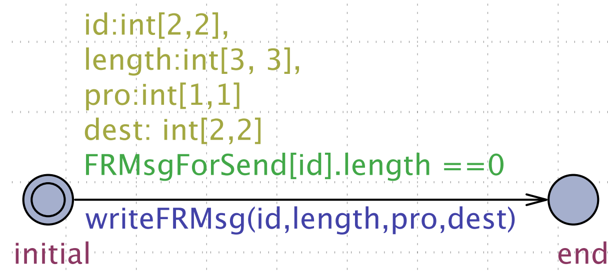


Figure 5.10: Task automaton for each E1 of the three IVN systems in *Case1*.

the FlexRay environment to the CAN environment. Hence, all response times are the same.

- For the backbone topology, we consider two CAN environments connected by the FlexRay protocol, so there are two forwarders between the CAN environments. Since `msg1` (`id=2`, `length=3`) must be forwarded twice to reach the destination environment, the response times of `msg1` are longer than for other topologies.

- In *Case 2*, we add one more environment node into IVN systems on each topology, as shown in Fig. 5.11. An application model is applied to release messages in each topology, as shown in Fig. 5.12. Here is a `msg1` (`id=2`, `length=3`) is sent from E1 to E3.

  - For the central and backbone topologies, even though there is one more environment than in *Case 1*, the transmitting process of `msg1` is unaffected. The

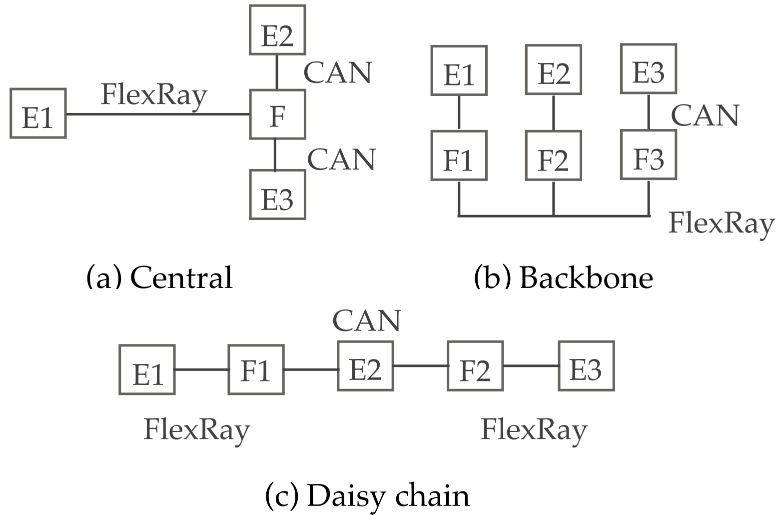


Figure 5.11: Topologies in *Case2* and *Case3*.

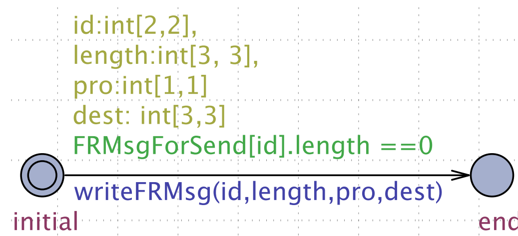


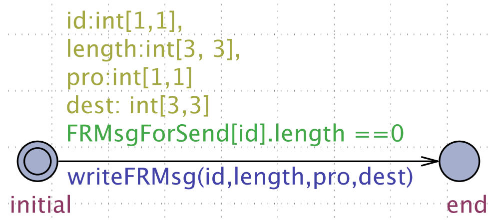
Figure 5.12: Task automaton for each E1 of the three IVN systems in *Case2*.

BCRT and WCRT values in *Case 2* are the same as in *Case 1*.

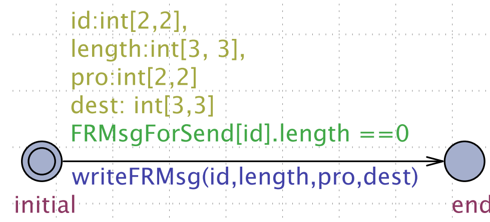
– The system model with daisy chain topology has a CAN environment E2 among E1 and E3, i.e. There are two forwarders to connect these three environments. As a result, from E1 to E3, the `msg1` goes through G1 and G2, the BCRT and WCRT values are greater than for *Case 1*.

- In *Case 3*, the structure is the same as for *Case 2* in the Fig. 5.11, but two messages are transmitted. There are two task models in E1 and E2, as shown in Fig. 5.13. `msg1` (`id=1`, `length=3`) is sent from E1 to E3; `msg2` (`id=2`, `length=3`) is sent from E2 to E3.

  – In the central and daisy chain topologies, `msg2` may trigger a forwarding delay in the forwarder because the bus is occupied by other messages. Therefore, the WCRT of the system in the central topology and daisy chain topology will increase, and the BCRT values remain the same as in *Case 2*.



(a) Task automaton in E1.



(a) Task automaton in E2.

Figure 5.13: Task automata for each E1 and E2 of the three IVN systems in *Case3*.

Table 5.5: Response time of each topology in the three cases.

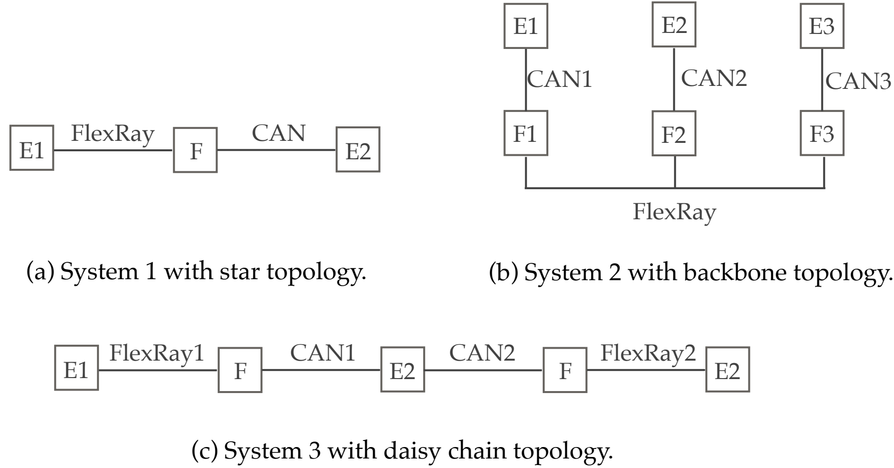
	Case 1		Case 2		Case 3	
Topology	BCRT	WCRT	BCRT	WCRT	BCRT	WCRT
Central	34	74	34	74	34	104
Backbone	65	106	65	106	65	106
Daisy chain	34	74	83	123	83	164

– In the backbone topology, the response time is unchanged. Because `msg1` and `msg2` can be transmitted in the same communication cycle of FlexRay part, the number of message identifiers does not impact of the response time.

Based on these results, we determined the impact of each topology on the message transmission process. When a lot of environments connect to a central forwarder, WCRT will be increased due to the efficiency of the gateway. If environments join the forwarder backbone of FlexRay, the response time will be affected only by the number of communication cycle. The number of environments between the sender and receiver in the daisy chain topology, impacts of response time. Since these results are consistent with the topological characteristics, we conclude that the framework can be applied to implement abstracted design models of IVN systems with different topologies.

5.3.3 Reusability

We depicted behaviors of each module in chapter 3 and chapter 4. In the framework, we defined the `<<Fixed>>` stereotype to describe *CAN*, *FlexRay*, *Interface*, *Medium* and *Configration* module. These module can be reused to build different IVN systems with little modification. We consider three IVN system with different number of environments,



(a) System 1 with star topology.

(b) System 2 with backbone topology.

(c) System 3 with daisy chain topology.

Figure 5.14: The topologies of three systems.

buses and topologies. These three systems have been introduced before, and theirs architectures are shown in Fig. 5.14. We will compare the models of the three system, and discuss the parts of the model that can be reused and need to be changed.

The *System1* has two environments and a forwarder, which is the IVN system design model described in section 4.6. The *System2* has three environments and three forwarders with backbone topology, which is the system with backbone in the *Case3* in section 5.3.2. The *System3* has three environments and two forwarders with daisy chain topology, which is the system with daisy chain in the *Case3* in section 5.3.2. We illustrate reusability by comparing source codes of the three system models in UPPAAL. Table 5.6 shows the number of code lines in each systems. First, we divide code of systems into four parts, *Configuration*, *CAN protocol*, *FlexRay protocol* and *Environments and forwarders*. Then, we calculate the amount of code in each part of the systems, and use the *System1* as a benchmark. After that, we compare code of each part between the systems.

- The *Configuration* segment of the source code is the declaration part for defining global variables. All systems have 103 lines. There are 8 lines of code in the *System2* and *System3* that are different from the *System1*, accounting for 7.7% of the *Configuration*, because we set different numbers of environments, protocols and forwarders.

- The *CAN protocol* segment has *CANInterface*, *Arbitration*, *Transceiver* and *CANMedium* that implement transmission behaviors and connection with environments and forwarders. In these three systems, they have the same code of the *CAN protocol*.

Table 5.6: Comparison of source code of the three systems.

Name	Number of Lines	Lines of source code in UPPAAL			
		Configuration	CAN protocol	FlexRay Protocol	Environments and forwarders
System1	Lines	103	139	483	138
System2	Lines	103	139	483	257
	Different Lines	8	0	0	-
	Percentage	7.7%	0%	0%	-
System3	Lines	103	139	483	254
	Different Lines	8	0	0	-
	Percentage	7.7%	0%	0%	-

- The *FlexRay protocol* segment has *FRInterface*, *FRMedium* and five automata, *POC*, *StaticMAC*, *DynamicMAC*, *NIT* and *FSP*, which implement transmission behaviors and connection with environments and forwarders. These three system have the same code of the *FlexRay protocol*.
- The *Environments and forwarders* segment is changeable in the framework. All systems have different designs, such as different topologies and environment designs. Therefore, we did not compare the reusability of this segment of the code.

For different IVN systems, we just need to build the *Environment* module and *Forwarder* module based on the system design, and then change parameters of the*Configuration* for setting other modules. In addition, we give a general description of the reusable parts and changeable parts of each module in the framework.

- *Configuration*:
  - Reusable: This module contains all parameters about other modules. These parameters describe the composition of an IVN system and set up protocols. These parameters apply to any abstracted IVN system operating with CAN and FlexRay. We do not need to change the name of these parameters, and can not add or subtract them.
  - Changeable: Each IVN system is described differently, such as the number of nodes, message length, protocol setting and so on. *Configuration* can describe

those information by changing the value of the parameters, and the value must meet constraint of the parameter described in the UML class diagram.

- *Environment*:
  - Reusable: This module presents two operations to write/read message to/from *Interface* for each CAN environment and FlexRay environment. These operations define behaviors of accessing *Interface*. They are fixed and can be reused in difference task models.
  - Changeable: *Environment* can describe a variety of functions to process messages. As we have shown in section 4.4, we can design different mechanisms for sending messages, such as the sending period, and add other functions to deal with messages. In addition, the number, identifier, length and destination of messages can be flexible changed in the task automaton, according with *Configuration*.
- *Forwarder*:
  - Reusable: This module models a scheduling algorithm to control message forwarding between CAN and FlexRay environments. There are four operations used to interact with *Interface* in Fig. 4.2. Thus, these operations can not be changed and can be reused to guarantee message transmissions.
  - Changeable: There are many kinds of gateways to handle message forwarding. In order to satisfy the diversity of gateways, the forwarding strategy of *Forwarder* can be established by users.
- *Interface*:
  - Reusable: This module is defined by CAN frame format and FlexRay frame format, and provides buffers of the frame format to store messages. The structure of the buffer is fixed for every node.
  - Changeable: The number and length of buffers are described by *Configuration* and changed by setting *Configuration* about messages.
- *CAN*:
  - Reusable: The *CAN* model is a part of *Communication Controller* module and has two automata. These two automata are reusable for any cases.
- *FlexRay*:

– Reusable: The *FlexRay* model is a part of *Protocol* module and has five automata. These automata are reusable for any cases.

5.3.4 Performance

To evaluate the performance of the framework, we use the simple example from section 4.1, to check the deadlock property, which is significant in exhaustively searching the state space of the system. The checking was conducted using UPPAAL 4.1.14 on a Mac OSX E1 Capitan machine with an Intel Core i7 3 GHz processor and 16 GB RAM. The memory usage and CPU times are listed in Table 5.7. We increase the number of message identifiers transmitted in the system. When the number of message identifiers is three, the checking result could not be obtained because of the explosion in the state space. Although we can only verify messages with few different identifiers, the number of the messages is huge. Since an environment is an abstracted subnetwork, and only performs tasks that require communication with other subnetworks, we defined a set of messages with a specific identifier to represent all messages that are sent from an environment to the other. Moreover, messages with a same identifier are sent repeatedly and model checking exhaustively checks all situations where messages are sent with different delays. Therefore, the framework is effective for verifying message communications between several environments. If there are many environments and messages, the capability of the framework is limited.

For qualitative verification, communication behaviors of the framework are verified corresponding to the CAN and FlexRay specifications. The framework is able to construct IVN systems with three kinds of topologies. Using the framework, the BCRT and WCRT of messages are checked in the IVN design models. We also verify the reachability of messages using two cyclic task models. The advantage of the framework is precise checking the transmission process and response time of messages, and is reusable for miscellaneous IVN system designs. However, the performance is insufficient to check a large IVN system with major message identifiers. Even if the checking results are accurate, there are not enough information to analyze the properties of the IVN system. For example, there is how many messages has been received within the BCRT and WCRT, and how many messages has the response time that is satisfied a certain time. In addition, IVN systems have a mass of indeterminate behaviors and probabilistic events, such as failures and control functions. For the purpose of solving such problems, we conceive statistical model checking for verifying quantitative properties of IVN systems.

To compare the performance of model checking and statistic model checking, we implement a probabilistic application model and a general application model with CAN

Table 5.7: Performance of qualitative verification

No. of message identifiers	Time (s)	Memory (kb)
1	0.181	11,304
2	63.755	604,768
3		Out of memory

Table 5.8: Performance of quantitative verification

	Time (s)		Memory (kb)	
No. of message identifiers	MC	SMC	MC	SMC
5	0.17	0.07	92,788	5,988
10	64.71	0.12	369,928	6,524
15	-	0.18	-	7,272
20	-	0.26	-	8,380

protocol. We examine a property in both cases to see if all messages with `id=1` could be received, and compare verification time and memory. As the number of the message identifiers increase, the traditional model checking becomes more and more difficult to obtain the result in a short time. Statistical model checking can easily check large amounts of message identifiers, and the verification time and memory increase linearly and slowly, as shown in Table 5.8.

For the quantitative verification, we add probabilistic behaviors in the application model. we have check probability of the reachability and probability density distribution of the response time using the same framework. Tasks of the application models write messages to *Interface* at a probability, and these messages will be received in a statistical frequency. From the probability density distribution, we can know that response time of message is concentrated in some intervals. Moreover, SMC is more efficient than traditional model checking. The advantage of the framework supports qualitative verification as well as quantitative verification. Whereas the checking results are vague, properties are satisfied with a probability. Hence, the qualitative verification and quantitative verification are complementary.

Chapter 6

Related Work

6.1 Verification of IVN Systems Based on Integration Platforms

In automobile industry, integration platforms are one solution for the IVN system design process [50, 2]. For example, DaimlerChrysler laboratory developed a tool including software and hardware architectures to flexibly construct and test IVN systems [1]; T. Demmeler et al. proposed a virtual integration platform to simulate and estimate the performance of IVN communication models [2]; H.Moon et al. implemented a heating ventilation and air-condition control system based on AUTOSAR architecture and tested it using MATLAB and SIMULINK [3]. The systems they examined were detailed and contained both software and hardware information, but they lacked a complete set of test cases, and a precise property specification.

There are other studies that are directly implemented an IVN system on integrated components and simulate and test electronic signal on hardware. For example, G. Feng et al. implemented a CAN system with electronic nodes and tested the system [4]; F. Baronti et al. had designed and implemented a FlexRay protocol component to verify fault tolerance of IVN systems [5]. These studies implemented the IVN system on physical devices, and then connected to PC for testing based on software platform. This method requires both software and hardware support, and the testing is based on electrical signals on hardware. The test effectiveness is directly dependent on test cases. Although these studies are closer to a real system, they lack completeness and flexibility.

Some studies only test and analyze the system itself. For instance, S. Anssi et al. focused on analyzing scheduling capability of AUTOSAR system [6]. Although these studies can be used to analyze and test IVN systems concretely and intuitively, a completed set

of test cases is difficult to design for checking properties of the system. Also, it is hard to precisely check concurrent behaviors and logic errors in the system design phase.

Compared with these studies, although our model has no operating system or physical protocol, this highly abstract model can reflect the communication behavior of the system in the system design phase and verify timed properties accurately and completely. Our proposed framework has the flexibility to build IVN systems with different protocols and topologies. Other researchers take multi-protocol into account, but they focus on improving the performance of gateways with CAN, LIN, FlexRay and MOST [49, 46, 51, 52]. The common idea is providing efficient scheduling algorithms to the gateway. For example, Kim et al. examines the implementation of a CAN–FlexRay gateway using message-mapping [46]. In our work, the gateway is used to forward messages, which only has the most basic function of forwarding messages and a simple scheduling mechanism. Our focus is on the effect of multiple protocols on timed properties. Although gateways can affect the communication efficiency of the system, even without gateways, the message transmission time between multiple protocols is different from a partitioned system, as mentioned the example in Chapter 1. Thus, our work verified the timed property effectively.

6.2 Model Checking of IVN Systems

Model checking is another effective method on verification of safety-critical systems. This method is exhaustively and automatically to check properties by searching all states of the system. The properties can be specified in temporal logic and verified precisely. There are many works on checking communication protocols [7, 8, 9]. They modeled protocols and verified properties from their specifications, such as the fault-tolerance on the FlexRay physical layer, the start-up process of FlexRay, and the error handling and fault-tolerance of timed-triggered CAN. These work focus on analyzing an verifying the protocol itself. Our work is to verify the IVN system design with abstracted applications.

For the software, some works [10, 11, 12, 13] aimed at verifying that implementations of the system is consistent with software standards. J. Chen et al. designed a model based on OSEK/VDX operating system and verified that the model meets OSEK specification by SPIN, and generated test cases using the model [11]. L. Fang et al. proposed a formal model of AUTOSAR multicore real-time operating system and developed a test case generator and a test program generator to complete system testing [13]. Y. Huang et al. implemented a formal model of OSEK/VDX OS with CSP language and verified the model using PAT [12]. They developed formal models of the operating systems and

the models are used to generate exhaustive test cases for helping system test. These studies did not consider communication protocols, but they analyzed task behaviors in the application layer. We do not take task scheduling in the application model, and tasks are simplified as a message distributor, which send messages in different cycles and probabilities.

In addition, some studies focus on interactive behaviors between ECUs in a single protocol [14, 15]. L.Waszniowski et al. established a whole CAN system model with OSEK/VDK operating system and showed a case study to verify timed properties of the system [14]. C.Pan et al. showed a CAN protocol model to verify some primary properties using UPPAAL model checker, and considered an application model with a scheduling algorithm to fix some properties that are unsatisfied [15]. Our previous work only modeled and verified communication behaviors of a FlexRay system [53]. However, an IVN system is extremely complicated, which contains multiple protocols, several gateways, and many applications. It is difficult to directly model the complete IVN system and efficiently verify properties. Moreover, the capability of verification is usually limited because the complexity of the system cause the state space explosion. To improve the capability and efficiency, we tried SMC techniques and changed the application model with probability.

6.3 Statistical Model Checking of IVN Systems

For statistical model checking, most works focus on the wireless system network(WSN) that are non-deterministic systems. They verified LMAC and CSMA/CA protocols, and analyzed throughout and delay with some probabilities. [40] considered stochastic events that can occur during the execution of CSMA/CA protocol, such as failures to access the medium, collisions and messages lose. [54] verified the probability of the clock synchronization of nodes with the different topologies. [55] presented a Timing-sync Protocol for Sensor Networks(TPSN) model with probability. This work verified the correctness of the model, and showed the performance of TPSN. [56] presents an arbitrary 5-node networks and analyze the router requests and entries with SMC-UPPAAL. These works have verified networks with stochastic protocols. In our work, CAN and FlexRay are deterministic protocols, and their behaviors are fixed. We used application models with probability that connect to protocol models, and checked quantitative properties. Our framework can carries out traditional model checking and statistical model checking in the same time.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

IVN systems are distributed real-time systems, which control driving safety, comfort and entertainment in a car, even driving assistance and automatic driving. A various of control systems combine to a IVN system. These control systems employ different communication protocols, since the cost, requirements, functions and other factors are considered. For an IVN system without a unified communication protocol, it is extremely complex and it is hard to precisely verify properties. From the example in section 2.6.2, we can see that different communication protocols affect data transmission. Furthermore, the IVN system is complicated and massive, which has a lot of stochastic behaviors. Consequently, in order to verify the IVN system accurately and effectively, we have presented a framework for verifying IVN system based on model checking techniques.

This thesis described a framework for modeling and verifying response times of messages in the IVN system that operate the CAN protocol and FlexRay protocol. The contribution of this work is describing an appropriate abstraction for modeling IVN systems and realize the associated framework; the framework is applicable and reusable for various IVN systems; qualitative verification and quantitative verification are applied to checking the IVN system design models based on the framework.

- Firstly, the IVN system needs to be abstracted through a two-stage abstraction on the basis of the protocol specifications of CAN and FlexRay, while preserving the necessary functionalities of communication.
    - An IVN system consists of hundreds of nodes, but not all nodes communicate with other subsystems in different communication protocols. This work is centered on the communication between different protocols, and ignores internal

communications within a subsystem. Hence, we abstracted the structure of IVN systems. That is, subsystem connected to a gateway is abstracted into an environment node. This environment following CAN or FlexRay communicates with other abstracted nodes through the gateway. We have simplified the structure of IVN systems and reduced communication traffic.

  - Environments are defined by CAN and FlexRay specifications. The specifications state the layer structure of CAN and FlexRay protocols, and expound functions of each layer. There are many descriptions related to lower part system design, such as encoding/decoding and bit signals on physical layer. This work is centered on analyze communication and time-related properties in the case of normal communication. Thus, we neglected the physical layer and fault handling in each protocol. Then, we showed the abstracted common structure of the protocols, simplified the functionality within them.

- Secondly, on the basis of the abstraction, we proposed a framework that formalized by a UML class diagram. The framework consists of *Environmnet*, *Forwarder*, *Interface*, *Communication controller*, *Medium* and *Configuration* modules which is established in UPPAAL model checker. The *Interface*, *Communication controller*, *Medium* and *Configuration* are fixed and reusable for building IVN system design models, which are constituted corresponding to the abstraction of CAN and FlexRay specifications. The *Interface* represents *Object* layer in ISO model, which is responsible for storing messages that need to be transmitted/received. It is the interface between *Environment* or *Forwarder*, and *Communication controller*, and the configuration of the *Environment* determines the connection between environments and protocols. The *Communication controller* includes two *Transfer* layers of CAN and FlexRay protocols. They control transmission process that conforms to communication schemes defined in the specifications. The *Configuration* is a set of parameters that account the composition, messages and protocols of an IVN system. These parameters can be modified for describing different IVN systems or improving IVN system designs. The *Environment* and *Forwarder* are changeable based on IVN system designs, because they are not defined by the specifications. They execute special functions embedded on ECUs, which are the *Application* layers of nodes in ISO model. We disregard data processing operations of these tasks and focus on sending and receiving data. The *Environment* simulates tasks that write message to *Interface* and request it to be send. The *Forwarder* performs scheduling and forwarding messages, which are modeled based on the gateway design. Thence, the framework is capable for modeling IVN design systems by changing *Environment*,

Forwarder, and parameter values of *Configuration*.

- In the experiments, the framework was shown to verify reachability and timed properties of message with abstracted IVN system models. In addition, the framework supports both qualitative verification and quantitative verification.

  - Using model checking, we did qualitative verification with deterministic *Environment* models. We precisely verified the validity of the framework, and response time and reachability of messages.
  - Using statistical model checking, we did quantitative verification with probabilistic *Environment* models. We checked the reachability of messages that are satisfied with a confidence degree, and the response time of messages are shown by probability density distribution plots.

- In the end, the framework has been evaluated from four aspects. At first, we have discussed the validity of the abstraction. The framework was proposed based on the two-stage abstraction. In the first stage, we abstracted the architecture of IVN systems, where subnetworks are abstracted as environments. Thus, we restored the subnetworks with different topologies and checked the response time of messages. The results suggest that the abstraction can preserve the timed properties of outside of subnetworks in the IVN system. In the second stage, we abstracted the CAN and FlexRay specifications. We listed some properties about transmission schemes that can be preserved, and the signal-level behaviors and error handling that can not be preserved in the framework. Then, the framework was applicable and reusable for different IVN system designs. We implemented IVN systems with three typical topologies and checked timed properties in the three cases. The checking results accorded with the characteristic of the topologies; for different IVN system design, the modules of the framework were unchanged, except for the *Environment* and *Forwarder* modules. Besides, the performance of the framework has been evaluated by model checking and statistical model checking. Model checking can exhaustively verify properties and give out precise results, but the capability and efficiency are limited; statistical model checking has better capability and efficiency, but the properties are satisfied with some degree of confidence. The framework can be used to both methods and their verification results can complement each other.

7.2 Future Work

We intend to extend the framework to verify IVN systems more practically. There are several directions for our future works.

- The quantitative verification did not take FlexRay system into account in this time. It is not sufficient to consider CAN system alone. We will take both CAN and FlexRay model into IVN systems to do quantitative verification.
- The IVN system is as a safety-critical system. Fault handling is indispensable to insure safety of the IVN system. We think to add the fault handling function into our framework, as a part of protocol models.
- If we extend the framework with the fault handling function, the performance of ordinary model checking will still be limited by state space problem. Besides, such system errors, message errors and failures in the IVN system, they can be represented by stochastic events. Thus, we will adopt probabilistic model to describe fault handling behaviors and verify IVN systems using statistical model checking.
- To enable the framework to verify more IVN systems, commonly used communication protocols will be abstracted as other protocol model in the framework, such as LIN, CAN-FD and Ethernet.

References

[1] K. Grimm, “Software technology in an automotive company - major challenges,” *25th International Conference on Software Engineering, 2003. Proceedings.*, vol. 6, pp. 498–503, 2003.

[2] T. Demmeler and P. Giusto, “A Universal Communication Model for an Automotive System Integration Platform,” pp. 47–54, 2001.

[3] H. Moon, G. Kim, Y. Kim, S. Shin, K. Kim, and S. Im, “Automation test method for automotive embedded software based on autosar,” *4th International Conference on Software Engineering Advances, ICSEA 2009, Includes SEDES 2009: Simposio para Estudantes de Doutoramento em Engenharia de Software*, pp. 158–162, 2009.

[4] G. S. Feng, W. Zhang, S. M. Jia, and H. S. Wu, “CAN bus application in automotive network control,” *2010 International Conference on Measuring Technology and Mechatronics Automation, ICMTMA 2010*, vol. 1, pp. 779–782, 2010.

[5] F. Baronti, E. Petri, S. Saponara, L. Fanucci, R. Roncella, and R. Saletti, “Design and verification of hardware building blocks for high-speed and fault-tolerant in-vehicle networks,” *IEEE Transactions on Industrial Electronics*, vol. 58, no. 3, pp. 792–801, 2011.

[6] S. Anssi, S. Tucci-Pergiovanni, S. Kuntz, S. Gérard, and F. Terrier, “Enabling scheduling analysis for AUTOSAR systems,” *Proceedings - 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2011*, pp. 152–159, 2011.

[7] M. Gerke, R. Ehlers, B. Finkbeiner, and H. Peter, “Model checking the FlexRay physical layer protocol,” *Formal Methods for Industrial ...*, pp. 132–147, 2010.

[8] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, “Verification of Flexray Start-Up Mechanism by Timed Automata,” *Proceedings of the 7th ACM & IEEE international conference on Embedded software - EMSOFT ’07*, p. 21, 2007.

[9] I. S. I. Saha and S. R. S. Roy, “A Finite State Analysis of Time-Triggered CAN (TTCAN) Protocol Using Spin,” in *Computing: Theory and Applications (ICCTA’07)*, 2007.

[10] L. Waszniowski and Z. Hanzálek, "Formal verification of multitasking applications based on timed automata model," *Real-Time Systems*, vol. 38, no. 1, pp. 39–65, 2008.

[11] J. Chen and T. Aoki, "Conformance testing for OSEK/VDX operating system using model checking," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pp. 274–281, 2011.

[12] Y. Huang, Y. Zhao, L. Zhu, Q. Li, H. Zhu, and J. Shi, "Modeling and verifying the code-level OSEK/VDX operating system with CSP," *Proceedings - 5th International Conference on Theoretical Aspects of Software Engineering, TASE 2011*, pp. 142–149, 2011.

[13] L. Fang, T. Kitamura, T. B. N. Do, and H. Ohsaki, "Formal model-based test for AUTOSAR multicore RTOS," in *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, pp. 251–259, 2012.

[14] L. Waszniowski, J. Krákora, and Z. Hanzálek, "Case Study on Distributed and Fault Tolerant System Modeling Based on Timed Automata," *Journal of Systems and Software*, vol. 82, no. 10, pp. 1678–1694, 2009.

[15] C. Pan, J. Guo, L. Zhu, J. Shi, H. Zhu, and X. Zhou, "Modeling and verification of CAN bus with application layer using UPPAAL," *Electronic Notes in Theoretical Computer Science*, vol. 309, pp. 31–49, 2014.

[16] W. Granig, D. Hammerschmidt, and H. Zangl, "Calculation of Failure Detection Probability on Safety Mechanisms of Correlated Sensor Signals According to ISO 26262," *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, vol. 10, no. 1, 2017.

[17] J. Wang, Y. Wang, C. Bi, J. Weng, and X. Yan, "Modeling the probability of freeway lane-changing collision occurrence considering intervehicle interaction," *Traffic Injury Prevention*, vol. 17, no. 2, pp. 181–187, 2016.

[18] A. D. Dominguez-Garcia, J. G. Kassakian, and J. E. Schindall, "Reliability evaluation of the power supply of an electrical power net for safety-relevant applications," *Reliability Engineering and System Safety*, vol. 91, no. 5, pp. 505–514, 2006.

[19] G. Agha and K. Palmskog, "A Survey of Statistical Model Checking," *ACM Transactions on Modeling and Computer Simulation*, vol. 28, no. 1, pp. 1–39, 2018.

[20] O. G. D. A. Edmund M.Clarke, Jr., "Model Checking," 2000.

[21] E. M. Clarke, "The Birth of Model Checking BT - link.springer.com," *Link.Springer.Com*, vol. 5000, no. Chapter 1, pp. 1–26, 2008.

[22] I. Romanovsky, "Model-checking real-time concurrent systems," in *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, p. 439, nov. 2001.

[23] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401–424, 1994.

[24] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Kluwer Academic, 1993.

[25] R. Bryant, "Graph based algorithms for {B}oolean function manipulation," *IEEE Trans. Computers*, vol. 358, no. 8, pp. 677–691, 1986.

[26] P. Godefroid and D. Pirottin, "Refining dependencies improves partial-order verification methods (extended abstract)," no. 6021, pp. 438–449, 1993.

[27] D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, vol. 8, no. 1, pp. 39–64, 1996.

[28] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Advances in Computers, Academic Press*, vol. 58, no. 99, pp. 117–148, 2003.

[29] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computations.* 2006.

[30] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pp. 414–425, 1990.

[31] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[32] R. Alur, "Timed automata," *Computer Aided Verification*, pp. 8–22, 1999.

[33] G. Behrmann, A. David, and K. Larsen, *A Tutorial on UPPAAL 4.0*, vol. 3185. 2006.

[34] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[35] P. Pop, P. Eles, Z. Peng, and W. Yi, "UPPAAL in a nutshell. Int.Journal on Software Tools for Technology Transfer," pp. 1(1–2):134–152, October 1997.

[36] D. R Alur, C.Courcoubetis, "Model-checking for realtime systems," *5th Symposium on Logic in Computer Science*, vol. 126, pp. 414–425, 1990.

[37] A. David, K. G. Larsen, A. Legay, M. Mikŭcionis, and D. B. Poulsen, "UPPAAL SMC tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, 2015.

[38] P. Bulychev, A. David, K. G. Larsen, M. Mikučionis, D. Bøgsted Poulsen, A. Legay, and Z. Wang, "UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata," *Electronic Proceedings in Theoretical Computer Science*, vol. 85, no. Qapl, pp. 1–16, 2012.

[39] P. Bulychev, A. David, K. Guldstrand Larsen, A. Legay, G. Li, D. Bøgsted Poulsen, and A. Stainer, "Monitor-based statistical model checking for weighted metric temporal logic," in *Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2012. Lecture Notes in Computer Science*, vol. 7180, pp. 168–182, 2012.

[40] Z. Hmidi, L. Kahloul, S. Benhazrallah, and C. Othmane, "Statistical Model Checking of CSMA / CA in WSNs," in *VECoS*, 2008.

[41] H. Heinecke, "Automotive system design - challenges and potential," in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 656 – 657 Vol. 1, march 2005.

[42] Bosch, "CAN Specification Version 2.0," 1991.

[43] K. H. Johansson and T. Martin, "Vehicle Applications of Controller Area Network," *Sensors Peterborough NH*, vol. VI, pp. 741–765, 2005.

[44] H. S. Seo and B. KIM, "Design and Implementation of a UPNP-CAN Gateway for Automotive Environments," *International Journal of Automotive Technology*, vol. 14, no. 1, pp. 91–99, 2013.

[45] Altran Technologies, "FlexRay Communications System Protocols Specification V.3.0.1," 2005.

[46] M.-H. Kim, S. Lee, and K.-C. Lee, "Performance Evaluation of Node-mapping-based Flexray-CAN Gateway for in-vehicle Networking System," *Intelligent Automation & Soft Computing*, vol. 21, no. 2, pp. 251–263, 2015.

[47] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in Automotive Communication Systems," *Proceedings of the IEEE*, vol. 93, pp. 1204 –1223, june 2005.

[48] M. Gerke, R. Ehlers, B. Finkbeiner, and H.-J. Peter, "Model Checking the Flexray Physical Layer Protocol," in *Formal Methods for Industrial Critical Systems*, pp. 132–147, 2010.

[49] K. Seung-Han, S. Suk-Hyun, K. Jin-Ho, M. Tae-Yoon, S. Chang-Wan, H. Sung-Ho, and J. Jae Wook, "A Gateway System for an Automotive System: LIN, CAN, and FlexRay," in *Industrial Informatics (INDIN'08)*, pp. 967–972, 2008.

[50] A. Sangiovanni-Vincentelli, "Electronic-system design in the automobile industry," *IEEE Micro*, vol. 23, no. 3, pp. 8–18, 2003.

[51] E. G. Schmidt, M. Alkan, K. Schmidt, and U. Karakaya, "Performance evaluation of FlexRay/CAN networks interconnected by a gateway," in *International Symposium on Industrial Embedded Systems (SIES)*, pp. 209–212, 2010.

[52] R. Zhao, G. H. Qin, and J. Q. Liu, "Gateway system for CAN and FlexRay in automotive ECU networks," *International Conference on Information, Networking and Automation (ICINA 2010)*, vol. 2, pp. 49–53, 2010.

[53] X. Guo, H.-H. Lin, K. Yatake, and T. Aoki, "An UPPAAL Framework for Model Checking Automative Systems with FlexRay Protocol," in *Formal Techniques for Safety-Critical Systems (FTSCS'13)*, vol. 419, pp. 36–53, 2013.

[54] L. Battisti, D. Macedonio, and M. Merro, "Statistical Model Checking of a Clock Synchronization Protocol for Sensor Networks," in *International Conference on Fundamentals of Software Engineering (FSEN)*, pp. 168–182, 2013.

[55] F. Zhang, L. Bu, L. Wang, J. Zhao, X. Chen, T. Zhang, and X. Li, "Modeling and Evaluation of Wireless Sensor Network Protocols by Stochastic Timed Automata," *Electronic Notes in Theoretical Computer Science*, vol. 296, pp. 261–277, 2013.

[56] A. D. Corso, D. Macedonio, and M. Merro, "Statistical Model Checking of Ad Hoc Routing Protocols in Lossy Grid Networks," in *NASA Formal Methods Symposium*, vol. 9058, pp. 112–126, 2015.