Model Checking of FlexRay Communication Protocol

By Xiaoyun Guo

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Toshiaki Aoki

September, 2012

Model Checking of FlexRay Communication Protocol

By Xiaoyun Guo (1010230)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Toshiaki Aoki

and approved by
Associate Professor Toshiaki Aoki
Professor Kokichi Futatsugi
Associate Professor Kazuhiro Ogata

August, 2012 (Submitted)

Abstract

Nowadays, the automotive systems mainly adopt electronic control units (ECUs) to realize X-by-wire technology. With the X-by-wire technology, the requirements or functionalities which was not able be realized mechanically before are now becoming possible. Generally, ECUs in an automotive system follow communication protocols to communicate with each other through one or multiple buses. Since communication protocols greatly affect the performance of an automotive system, protocols which can support high transmission rate and reliability are demanded. Recently, FlexRay communication protocol is considered as the de-facto standard of the automotive communication protocol. FlexRay communication protocol supports high transmission rate up to 10Mbs while still keeping the properties of reliability and fault-tolerance. These characteristics make FlexRay especially suitable for safety critical systems.

On the other hand, testing process of automotive systems is really time-consuming and complicated. In industry, devices implementing communication protocols should be prepared for testing applications implemented on ECUs. Connection between ECUs and protocol devices form a testing environment. Tests are conducted focusing on specific node or data transmission with support of data from previous tests. Since higher requirements result complex functionalities and therefore more ECUs are demanded, and the testing process becomes harder along with larger number of devices and high financial cost.

This thesis proposes a framework for verifying design model of automotive systems with FlexRay. The framework is based on UPPAAL, with a model checker of timed automata for modeling the time-related behavior. By using the framework for verifying the design models, developers could have better chances to find bugs at the design level with the support of model checking technology. This may lead to a reduction of cost of the system development while increasing the quality of applications.

Similar to the devices-based testing, the UPPAAL framework consists of a FlexRay model and an application model. The former one represents the FlexRay devices and the later represents the ECUs where applications are implemented. Since UPPAAL only provides primitive synchronization using channels, the FlexRay model and the application model have to be specially handled.

The FlexRay model is built upon the specification with three steps of abstraction:

- **Essential Component Selection:** The model design verification of applications only requires functionalities of sending and receiving frames to be presented in the FlexRay model. Therefore, only essential and necessary components which provide the communication functionalities are selected.
- **Functionality Reduction:** With components selected in first step, it is not necessary that the FlexRay model has the full behavior in sending and receiving frames. Behaviors such as adjustments for error control (i.e. fault tolerance feature) can be skipped.

- **State Space Reduction:** Due to the heaviness of the FlexRay model after step one and two, further abstraction is conducted in order to reduce the state space while preserving functionalities implemented.

The FlexRay model also provides parameters and interfaces for the communication and access of the application model. The application model only needs to follow these parameters and interfaces to cooperate with the FlexRay model as an automotive system design model.

To evaluate the framework, experiments are conducted as follows: (1) A testing application models with basic behaviors are introduced for examining the validity of the FlexRay model; (2)A simple application is established to verify the response time of the system which is tested by using observer model; (3)A simplified adaptive cruise control system is introduced to show the feasibility by using the framework on verifying the practical applications. The results of the experiments prove the validity of the FlexRay model and the feasibility of the framework, respectively. Timing properties are especially examined and experiences of improving the application model from checking results are learned.

Acknowledgments

First of all, I would like to show my deepest gratitude to my supervisor, Assoc. Prof. Toshiaki Aoki for his guidance, advice, assistance and support during my Master study.

I would like to express my sincere thanks to Prof. Kokichi Futatsugi for his encouragement and helpful comments.

Furthermore, I have to thank all the members in both Aoki Laboratory and Futatsugi Laboratory for their kindly helps to me not only in the research but also in the daily life.

I would like to thank Shinhong Lin for giving me so much help in completing this work.

Finally, I would like to thank my parents for encouraging and supporting me during my whole study in Japan, and my boyfriend Meng Cheng for his accompanying, supporting, and understanding.

Contents

Abstract **1**

Acknowledgments **3**

List of Figures **6**

List of Tables **8**

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Objective . . . 10
1.3 Related Works . . . 10
1.4 Overview of Proposed Approach . . . 11
1.5 Structure of the Thesis . . . 11

2 Background **13**
2.1 Automotive System . . . 13
2.2 FlexRay . . . 14
2.3 Model Checking . . . 15
2.4 Timed Automaton . . . 17
2.5 UPPAAL . . . 18
2.5.1 Timed Automaton in UPPAAL . . . 18
2.5.2 The UPPAAL Modeling and Verification . . . 19
2.5.3 A Simple Example . . . 20

3 Construction of FlexRay Model **23**
3.1 Overview of FlexRay Communication Protocol Specification . . . 23
3.2 Abstracted FleRay Model . . . 24
3.3 FlexRay Model . . . 26
3.3.1 Communication Cycle . . . 26
3.3.2 Protocol Operation Control . . . 29
3.3.3 Media Access Control . . . 31
3.3.4 Timmer . . . 39
3.3.5 Frame and Symbol Processing . . . 41

4 Model Checking Applications with FlexRay Model **43**
4.1 Model Checking Applications with FlexRay Model Process 43
4.2 Interface of FlexRay Model 44
4.3 An Example of Communication System with FlexRay Model 45
4.3.1 Application Model 45
4.3.2 An Example of Communication System 47

5 Experiments **49**
5.1 A Testing Example 49
5.1.1 Verification 50
5.1.2 Evaluation 52
5.2 Response Time Checking with FlexRay Model 52
5.2.1 Verification 54
5.2.2 Evaluation 55
5.3 Adaptive Cruise Control Subsystem 55
5.3.1 Verification 57
5.3.2 Evaluation 58

6 Conclusion **60**
6.1 Summary 60
6.2 Future Work 60

List of Figures

1.1 Framework 11

2.1 Model checking method 16
2.2 A simple example of a timed automaton 17
2.3 (*a*) The train model and (*b*) the gate model in UPPAAL 21

3.1 The topology of FlexRay bus system 24
3.2 The structure of node 25
3.3 Simplified structure of node 26
3.4 Timing hierarchy within the communication cycle 26
3.5 Communication cycle of FlexRay model 27
3.6 FlexRay communication cycle example 29
3.7 Overview of protocol operation control 30
3.8 Protocol operation control model in UPPAAL 31
3.9 Media access process 32
3.10 Media access in static segment 33
3.11 Media access in dynamic segment 34
3.12 (*a*) action point a (*b*) action point b 35
3.13 Media access in dynamic segment arbitration 36
3.14 MAC static segment model in UPPAAL 37
3.15 MAC dynamic segment model in UPPAAL 38
3.16 Network idle time model in UPPAAL 39
3.17 Timer model in UPPAAL 40
3.18 Reception related events 41
3.19 Overview of frame and symbol processing 41
3.20 Frame and symbol processing model in UPPAAL 42

4.1 Flow chart of the checking model 44
4.2 (*a*) The sender model and (*b*) the receiver model in UPPAAL 48

5.1 Testing example model in UPPAAL 50
5.2 (*a*) ECU1 (*b*) ECU2 (*c*) ECU3 (*d*) ECU4 (*e*) Receiver 51
5.3 The structure of a simple application model 53
5.4 (*a*) The sender model and (*b*) the receiver model in UPPAAL 54
5.5 Observer model 54

5.6 Adaptive cruise control system 56
5.7 The communication cycle of adaptive control system 57
5.8 The ECU1 model of ACC 57
5.9 The ECU2 model of ACC 58
5.10 The ECU3 model of ACC 58

List of Tables

2.1 The query language of UPPAAL 20

4.1 Configuration parameters of interface 46

5.1 Results of the experiments 54
5.2 The relevant parameters value of the adaptive cruise system 56

Chapter 1

Introduction

1.1 Motivation

The automotive control system is mainly composed of electronic control units (ECUs), sensors and actuators. The sensors transmit data information to the corresponding ECUs, where the data will be processed and further transmitted to the actuator. After receiving the information, the actuator will give a response. Generally, the system function can be fulfilled by these procedures mentioned above. With the rapid development of the electronic technology and automotive industry, higher requirements are needed in term of safety, entertainment and comfortability, and more and more electronic components are installed in the vehicles. Therefore, the communications among components are getting more difficult and complicated, while the conventional communication mode is no longer satisfactory. As a consequence, more advanced communication protocols are desperately required, in order to sustain the robustness, safety, instantaneity and reliability of the automotive system.

Nowadays, control area network (CAN), local interconnect network (LIN), media oriented system transport (MOST) and FlexRay are commonly used in automotive industry [1]. Among those bus protocols mentioned above, CAN and LIN are the most popular ones. CAN is an event-triggered protocol used in the power system, which has the longest history in the global automotive industry, while LIN bus system is a low cost solution. MOST is mainly used in the network audio/video field as one of the additional functions of the automotive system. FlexRay is the most advanced communication standards, which is also the focusing point of this research. FlexRay is designed for safety-critical systems [2]. It is faster and more reliable than other protocols, allowing the event-triggered and time-triggered messages sharing the same bus. Actually, FlexRay is also a flexible hard real-time system, which can better meet the needs of the automotive system.

In order to ensure that the automotive system can be successfully applied in the vehicles, simulations and testings are necessary. The general simulating and testing methods employ hardware equipments to verify the bus communication such as [3] [4]. The interface tool is used to separate the application layer and the transport layer, and test ECUs and the bus, respectively. The protocol drivers and interfaces are provided by device

vendors, the majors of which are Qtronic and Vector. Conventionally, during the testing procedure, the graphical user interface (GUI) tools and the simulator are combined. Usually, Matlab is used as the simulator, as well as some other simulation devices. The GUI are used to check the raw data and the graphic. The configuration and the code implementation are also needed to fulfil the system function.

The system checking takes into account of the practical communication environment based on the hardware, such as noises, temperature and etc. However, this method has some limitations, because it is physical-layer-based and only focuses on signals and voltage, etc. The system functions are carried out by the code actuator. In other words, developers have to embed the software into ECUs. Then, the ECUs are connected to the FlexRay communication drive tool and check the devices, and the developer test and simulate the system. Moreover, due to the fact that this method partially tests the selected nodes, it requires a big amount of devices with high time and financial cost. In addition, it is hard to found bugs on the high-level from the low-level data, such as system functions and time-related properties. Therefore, a simpler and more convenient method is proposed to test the communication between ECUs in the primary stage of the automotive system design.

1.2 Objective

The main goal of this thesis is to propose an UPPAAL framework to test and verify the automotive systems using FlexRay protocol. First of all, a reusable FlexRay model is designed and established in UPPAAL. Secondly, another application model is simply designed according to the specification of the users. Then, the two models are connected together via the interface, forming an automotive communication system. By using UPPAAL, we finally check the function and time-related properties of communication system. One of the advantages of this approach is that the model checking method can provide exhaustive checking. Moreover, this framework checks properties from timing and functional aspects and allows modifications in the primary stage of development. It is less costly and does not need any hardware devices.

1.3 Related Works

In the recent years, significant progresses have been made on the research of the FlexRay communication system. On the one hand, the FlexRay system is analyzed in terms of timing and scheduling, from the prospective of the ECU. Such as in [5], [6] and [7], they usually calculate the worst case response times of all the tasks and messages in the FlexRay system. On the other hand, the research of FlexRay on the physical layer is did based on the physical protocols [8][9][10]. Additionally, other properties of FlexRay are verified. Such as the fault-tolerance analysis in [11] and [12], the author proposed a self-organized distributed coordinator concept which performs the self-reconfiguration in case of a node failure using redundant slots in the FlexRay schedule and combination of messages in

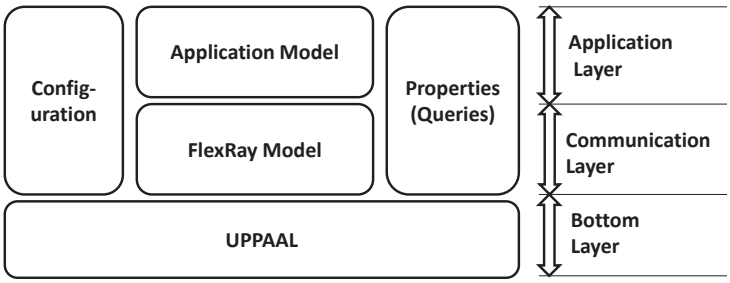


Figure 1.1: Framework

existing frames and slots to avoid a complete bus restart. The starting process is also discussed as in [13]. Although many aspects of the FlexRay system have been studied, the most important factor is time-sensitivity for a real-time system. On the basis of [5], we use timed automata theory for the verification. We focus on the communication between nodes in a FlexRay communication system, then model and simulate it. A verification framework is proposed to check the feasibility, instantaneity and reachability of the FlexRay communication system. Additionally, we propose a FlexRay model which can be applied in the practical development. Through the verification, developers can use the model to modify the design system in the early design stage.

1.4 Overview of Proposed Approach

In this work, we propose an UPPAAL framework for verifying and testing automotive systems using FlexRay protocol, where the structure is shown in Fig. 1.1. The bottom layer is the UPPAAL model checker as the foundation of the framework. All the activities are completed based on it. The communication layer is actually a FlexRay model, which is abstracted from FlexRay communication protocol. It is the core of the framework to achieve protocol service. Based on this part, the application layer is the application design model of the automotive control systems. In addition, the configuration module provides parameters of FlexRay model and application model. The queries are the properties in UPPAAL, which will be tested in the model checking part. It used to verify some properties between application layer and communication layer. By using the framework for verification of design models, developers could have better chances to find bugs at the design level with the support of model checking technology. This may lead to reduction of cost of development while increasing the quality of applications.

1.5 Structure of the Thesis

This thesis is organized as follows. The background of this work is introduced in Chapter 2. Then in Chapter 3, the overview of the proposed approach is presented and the

process of test generation is defined. Chapter 4 introduces the details of establishing FlexRay model in UPPAAL. Chapter 5 shows the way of building application model and interface of the FlexRay model, and discusses how to connect them together. In Chapter 6, the experimental results and evaluations are presented. Finally, Chapter 7 gives the conclusions, as well as the future work.

Chapter 2

Background

2.1 Automotive System

In modern automotive industry, machinery and hydraulic system are gradually superseded by X-by-Wire [12], which removes the machinery and hydraulic backup system, and is safety-related with certain fault-tolerant capacity. The traditional mechanical control components are partly replaced by ECU. With the development of the automotive electronic technologies, there are more and more electronic and electrical devices within one system, and mostly every module is controlled by a ECU and works independently. With the increase of the vehicle performances, more and more ECUs are equipped [14]. The conventional point-to-point communications will cause many difficulties when installing, checking and maintaining the system. All electronic components are connected by the bus as a network, which reduces the nodes and conducting wire, as well as enhances the reliability and flexibility of the system.

Recently, with the increase of active safety systems and the realization of intelligent driver assistance function, such as the Side Assist system and Lane Assist system, the traffic accidence can be reduced. The driving safety will be significantly promoted, but it also yields higher requirements of the system complexity, such as the determinacy of information.

For the safety-critical systems, they have to be strictly fault-tolerant and deterministic with broad bandwidth. The fault tolerance and determinacy play a key role in the system reliability, and broad bandwidth enables high-speed and more accurate data transmission. In this case, the delay time can be dramatically reduced, and the flexibility and safety will also be promoted. Therefore, complex car network technologies are rapidly developed and various types of bus are used in the automotive industry, such as CAN, LIN, MOST, FlexRay and so on.

As the most mature bus, CAN is widespread in electronic control systems and communication systems of the modern automotive industry, such as engine control system, automatic transmission control system, Anti-lock brake system (ABS), automatic cruise control (ACC) and vehicle-mounted multi-media system [15]. As the micro control communication bus of vehicles, CAN supports information exchanging among the electronic

control devices, which forms the automotive electronic control network. LIN bus is typically used in car doors, guide pulleys, seats, engines, climate control, lightings, rain sensors and Intelligent wipe devices. It is convenient to link those devices together into the automotive network to support the accesses of various of diagnosis and services. The common used analog signal coding can be superseded by digital signals for harness optimization [16]. MOST is mainly used in network audio/video fields, usually as an additional function of a car. In general, it has nothing to do with the vehicle control system. FlexRay can be used in different automotive fields, such as dynamical system chassis control system, active safety system and wire control drive system. It can also replace several high-speed CAN buses [17], which therefore reduces the complexity and the cost. The next evolution of the automotive network will be based on the cluster architecture, which employs FlexRay as the strong core.

The transmission rate, reliability and cost index of CAN are appropriate in the application of the current automotive power assembly. Due to that CAN is event-triggered, when more than one information streams are going to be transmitted at the same time, the traffic congestion will happen. If information is sent with different priorities, some of them will have large time delay and it is hard to estimate the delay properties of the information.

In the real-time system, such as X-by-Wire, the communication scheduling of the entire network has to guarantee that the information is able to be transmitted within a known time duration in arbitrary network conditions. However, it is difficult for the standard CAN to meet this requirement. FlexRay is time-triggered and fault-tolerant, with small and fixed time delay and a high speed bus. FlexRay can strictly meet the requirements of reliability, availability and conformance, as well as safety and dynamic control functions requirements for X-by-Wire system. The FlexRay bus system is introduced in detail as follows.

2.2 FlexRay

The FlexRay™ Communications System is designed in automotive applications which is robust, scalable, deterministic and fault-tolerant. It was first developed by the FlexRay™ Consortium, a cooperation of leading companies in the automotive industry, from the year 2000 to the year 2010. The FlexRay™ Consortium has concluded its work with the finalization of the FlexRay™ Communications System specifications Version 3.0.1[18]. FlexRay uses x-by-wire technique to reduce the reliability on the hydraulic system of the vehicle controls. FlexRay was first employed in 2006 in the pneumatic damping system of BMW's X5, and fully utilized in 2008 in the BMW 7 Series[19]. The FlexRay specification was completed in 2009 and is widely expected to become the future standard for the automotive industry[20] [10].

FlayRay has two separated buses, each with a transmission rate of 10 MBit/s. The information transmitted in the two buses are mainly the redundancy and the fault tolerance, when the system is with high validity requirement. In other cases, different information data can be transmitted on the buses, with the throughput being doubled. FlexRay can

also be used at low rate of 2.5 or 5 MBit/s, and it also defines the start, bus or the mixed topologies for data transmission. The access method used in FlexRay is based on the synchronization timing. The FlexRay provides services by automatically establishing the connection and timing synchronization. The accuracy of timing is between 0.5 μs and 10 μs, but usually 1-2 μs. By using a special algorithm, all the local clocks of the nodes are amended to be synchronized with the overall clocks. The synchronization information is transmitted during the stationary segment of the communication period.

Besides FlexRay also supports time-triggered and event-trigged communications. It follows the principles of Time Division Multiple Access (TDMA). In TDMA network, messages in communication channel are transmitted in different time slots. Each communication cycle is comprised of a certain amount of time slots. The slot number is counted from one in an ascending order until one communication cycle ends. For FlexRay, the nodes with the information data are allocated into the certain time slots, and they are able to uniquely get access to the bus during their own time slots. Due to the fact that the time duration of the information on the bus is predictable, the access to the bus is deterministic. However, one of the drawbacks is that, the bus bandwidth can not be fully used if it is stationarily allocated by determining the time slots of the nodes and information data. FlexRay divides one period into static and dynamic segment. While in the dynamic segment, the time slots are dynamically allocated, which is known as the flexible-TDMA principles. In each case, only within a short period of time the bus can be accessed (it is called minislots). If any access happens during a minislot, the time slot will be expanded according to the required time. Therefore, the bus bandwidth is dynamically changeable. It assures, on the one hand, the certainty of the bus access, and on the other hand, offset the inadequacy of the stationary transmission.

2.3 Model Checking

Model checking [21] is an automatic technique for formal verification of finite-state concurrent systems. It has a number of advantages over traditional approaches for this problem, proven by the simulation, testing, and deductive reasoning. Pioneering work in the model checking of temporal logic formulae was done by E. M. Clarke and E. A. Emerson and by J. P. Queille and J. Sifakis in the 1980s. At present, model checking has been applied to many fields, such as in computer hardware, communication protocol, control system, safety of authentication protocol. It has achieved remarkable successes, and also has been extended to industry from academia.

The basic idea of model checking is shown in Fig. 2.1. The state system indicates the behavior of the system. The temporal logic formula (F) describes the properties of the system. Whether the system satisfies the expectation of properties is transformed into math problem, $\{s \in S \mid M, s \models f\}$ [21].

Model checking is an efficient searching procedure of abstracting a system or procedure into a finite-state model. It is performed to automatically determine whether the model impacts the properties or not, by modeling the concurrent systems as finite-state automata and formulating the properties in temporal logic. If the state space searching shows that

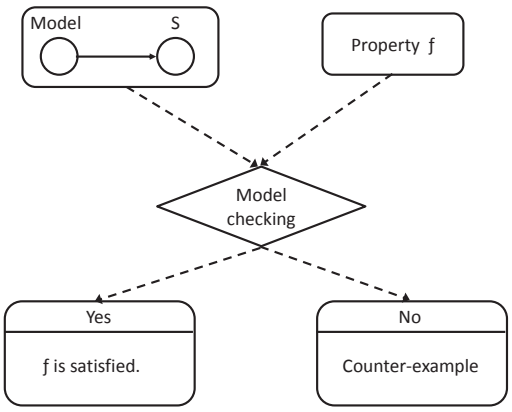


Figure 2.1: Model checking method

there is no violation, then the correctness with regard to the property is proved. If the property violation is detected, a counterexample will be generated to illustrate the property violation. However, it also includes the analysis results of the human assistance. Based on the analysis of the counterexamples, the system process can be modified and re-verified.

There seems to be many differences between the formal verification and testing. Specifically, formal verification is static which involves system models analyzing, covering all paths in the model. However, testing is dynamic and deals with the real-world system itself, which is its implementation or source code, but often covering only a limited number of system paths. Model checking is more strict and exhaustive, and especially more suitable for the verification of the large-scaled softwares and protocols, guaranteeing the reliability, correctness, consistency and integrality.

The model checking problem is easy to describe. In the process of model checking, the system or the process is transferred into a formalized and finite-state model, which can be accepted by model checking tools. For the large-scaled and complicated system, an abstract model has to be designed, omitting the uncorrelated and less important details. To test whether the design satisfies the requirements of the specification, the first step is to logically express these properties. The most difficult point is to make sure that all the properties are found to be satisfied with the specification.

The main challenge in model checking is to deal with the state space explosion problem. Due to the fact that model checking is based on the exhaustive searching of the system's state space, the state amount of the finite-state model exponentially increases as the concurrency grows[22], and therefore it is very difficult to construct the achievable state space in the computer memories, when establishing the complex models. In order to efficiently perform model checking, it is needed to reduce the compress the state space. There are several approaches to combat this problem. One of the proper ways is to relief the blow up problem by utilizing the structure properties of the state space of the model, such as symbolic model, symmetric model, partial order model and the On-the-fly model[23].

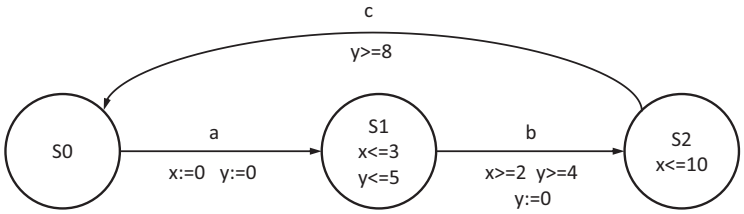


Figure 2.2: A simple example of a timed automaton

Another approach is to abstract or decompose the complicated model test. These approaches have been implemented in different model checking tools. One of the advantages of model checking is that it can completely and automatically execute verification, which is supported by the effective software tools. The most famous concurrency verification tools are SMV and SPIN, etc.

2.4 Timed Automaton

To cope with the problem of continuous time, the timed automata model was proposed by Alur, Courcoubetis and Dill, as an extension of the automata-theoretic approach of modeling the real-time systems [24]. Timed automaton introduces temporal constraint based on the conventional finite state machine. Assuming that the state transition is instantaneous, time lapse happens at a certain state. In the timed automata, clocks always start from 0 when the system begins, and they are all synchronized and increase with the equal rate. Clocks can be reset at arbitrary state transition point, with a time duration starting from the previous reset until now. For fear of errors, timed automaton is assumed to has finite times of transitions in the finite time duration. Time automation can be expressed by state transition diagram of the temporal constraint, based on the finite true-value clock variances [25]. In the diagram, peak points denote the locations, edges are the transitions, clock constraints are the guards, which are related with transitions. The transition can be taken only if the current values of the clocks satisfy the clock constraint. The clock constraints can be also related to the locations, called the invariant. Only when the invariant becomes true can time lapse happens at this location.

A simple example of a timed automaton is shown in Fig. 2.2. The automaton consists of three locations s_0, s_1 and s_2, two clocks x and y, and three transitions are a, b and c. The automaton starts in location s_0, and it can make an a transition to location s_1 and reset the clock x and y to 0. The automaton can remain in location s_1 as long as x is less than or equal to 3 and y is less than or equal to 5. When x is at least 2 and y is at least 5, it can make a b transition to location s_2 and reset clock y to 0. And the automaton remain in location s_2 on condition y is less than or equal to 10. While y is at least 8, it can make a c transition back to location s_0.

The theory provides a formal framework to model and analyze the behavior of real-time systems. The correct functioning of the real-time systems has to guarantee the strict

timing constraints such as execution times, response times, tasks periods, communication delays, etc.Many tools have been developed based on the timed automata, such as COSPAN, KRONOS[26], EPSILON, RT-SPIN and UPPAAL. In our research,we choose UPPAAL as model checking tool.

2.5 UPPAAL

UPPAAL is an integrated tool environment for modeling, simulation and verification of real time system modeled as network of timed automata [27]. It is jointly developed by Uppsala University and Aalborg University. Since UPPAAL first came out in 1995 [28], it has been continuously updated and the latest version is UPPAAL 4.1.11. It serves as a modeling or design language to describe system behavior as networks of automata extended with clock and data variables. UPPAAL is to verify the clock constraint and system accessibility by searching the state space of the system, with high efficiency and practicality. It has been applied successfully in case studies ranging from communication protocols to multimedia applications.

2.5.1 Timed Automaton in UPPAAL

UPPAAL is based on the theory of timed automata and its modeling language offers additional features such as bounded integer variables and urgency. In UPPAAL, a system is modeled as a network of several such timed automata in parallel. The following give the basic definitions of the syntax and semantics for the basic timed automaton [29].

Definition 1 (Timed Automaton (TA)) A timed automata is a tuple (L,l_0,C,A,E,I), where L is set of locations,$l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions and the internal τ-action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \to B(C)$ assigns invariants to locations.

Definition 2 (Semantics of TA) Let (L,l_0,C,A,E,I) be a timed automaton. The semantics is defined as a labelled transition system $< S, s_0 \to >$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$ is the transition relation such that:

-$(l, u) \xrightarrow{d} (l, u + d)\, if \forall d' : 0 \leqslant d' \leq d \Rightarrow u + d' \in I(l)$, and

-$(l, u) \xrightarrow{a} (l', u')$ if there exists $e = (l, a, g, r, l') \in E$ s.t. $u \in g$, $u' = [r \mapsto 0]\, u$, and $u' \in I(l')$,

where for $d \in \mathbb{R}_{\geq 0}$, $u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]\, u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over C/r.

Definition 3 (Semantics of a network of TA) Let $A_i = (L_i, l_i^0, C, A, E_i, I_i)$ be a network of n timed automata. Let $\bar{l}_0 = (l_1^0, ..., l_n^0)$ be the initial location vector. The semantics is defined as a transition system $< S, s_0 \to >$, where $S = (l_1 \times ... \times l_n) \times \mathbb{R}^C$ is the set of states, $s_0 = \left(\bar{l}_0, u_0\right)$ is the initial state, and $\to \subseteq S \times S$ is the transition relation defined by:

-$(\bar{l}, u) \xrightarrow{d} \left(\bar{l}, u + d\right) if \forall d' : 0 \leqslant d' \leq d \Rightarrow u + d' \in I(\bar{l})$.

-$(\bar{l}, u) \xrightarrow{a} (\bar{l}[l'_i/l_i], u')$ if there exist $l_i \overset{\tau g \tau}{\rightarrow} l'_i$ *s.t.* $u \subseteq g, u' = [r \rightarrow 0]u$ and $u' \in I(\bar{l}[l'_i/l_i])$.

-$(\bar{l}, u) \xrightarrow{a} (\bar{l}[l'_j/l_j, l'_i/l_i], u')$ if there exist $l_i \overset{c?g_i r_i}{\rightarrow} l'_i$ and $l_j \overset{c!g_i r_i}{\rightarrow} l'_j$ s.t. $u \in (g_i \wedge g_j), u' = [r_i \cup r_j \rightarrow 0]u$ and $u' \in I(\bar{l}[l'_j/l_j, l'_i/l_i])$.

2.5.2 The UPPAAL Modeling and Verification

UPPAAL is able to simulate the model behavior. It can simulate the execution process of the model step by step, and also automatically test if the model satisfies accessibility and give a trace of execution. The validity is determined by a series of property checking. The Java client graphical interface is comprised of 3 main parts: an Editor,a simulator and a verifier. The editor is used to create and edit the target system. Systems are represented by a network of timed automata put in parallel. By instantiating the parameters module, a system can add the clock constraint. Every system synchronizes with with each other by using shared variables and communication channels. Clocks and integer variances are expressed by 5 expressions as follows:

- **Guard**
  Guard condition defines a condition for transition between two locations. The transition can be executed only if the guard condition is fulfilled.
- **Synchronization**
  Synchronization serves for synchronization between timed automata. One of them is condidered to be in a location leading to another one by means of transition with synchronization parameter equal to name? is executed in another concurrently running timed automaton, the transition with sync = name! will be executed in automaton A too.
- **Update**
  Update allows initialization of variables including the clock. The updating is performed when transition is executed.
- **Invariant**
  Invariant condition defines a condition expressing an urgency of leaving the location. If the invariant condition is satisfied, the system can stay in or leave the location. The location has to be abandoned before the invariant condition can be said to be unsatisfied. If the system is not able to do it because of guard conditions for all possible transitions and invariant condition in appropriate location are unfulfilled at the same time, there is a deadlock in the system.
- **Selection**
  Update allows initialization of variables including the clock. The updating is performed when transition is executed.

Besides the normal location, committed and urgent locations are added in UPPAAL as the improvement. At the urgent location, time may not elapse, all clocks with regard to transitions at the urgent location will be reset. At committed location, the execution of time procedure must not be interrupted and the execution procedure does not consume

Name	Property	Equivalent to
Possibly	E<>p	
Invariantly	A[ ]p	not E<> not p
Potentially always	E [ ] p	
Eventually	A<>p	not E [ ] not p
Leads to	p→q	A[ ] (p imply A <> q)

Table 2.1: The query language of UPPAAL

time. In addition, UPPAAL provides both the broadcast channel and the urgent synchronization channel. The broadcast channel is declared by broadcast *chan* c, and one sender can synchronize with arbitrary number of receivers, without any congestion. The urgent synchronization channel is declared by urgent *chan* c, whose synchronous transition is without time-delay. There must not be clock constraint at the edge of synchronous transitions. After being established at the editor, the system model is checked in the simulator, in order to find errors before starting verification. On the other hand, users can import the trace obtained during the verification, and test the system through the variable view, the system view and the message sequence chart.

Verifier is used to test whether the system meet the requirements of the spectation. The query language written in verifier is used to specify properties to be checked. It is a subset of Timed Computation Tree Logic (TCTL) [30]. The query language consists of path formulae and state formulae. State formula is an expression that can be evaluated for a state without looking at the behavior of the model. Path formula is an expression that describes the behavior of a path in the model.

To test the reachability, safety and liveness properties, UPPAAL provides 5 verification languages, which can be summarised as follows Tab. 2.1:

- $E <> p$ implies the *possibility*. $E <> p$ is true if and only if there is a sequence $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n$ in the timed automata, where s_0 is the starting state and s_n is p.
- $A[]p$ is *Invariantly*, which is equal to $E <>$ not p.
- $E[]p$ means *Potentially always*. In the timed automata, when $E[]p$ is true and if and only if there is a sequence $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_i \rightarrow ...$ where p is satisfied for all state s_i, and this sequence is infinite or stopped at state (l_n, v_n).
- $A <> p$ means *Eventually*, which is equal to not $E[]$ not p.
- $p \rightarrow q$ denotes *Leads to*, which is identical to $A[]$ (p implies $A <> q$).

2.5.3 A Simple Example

In this section, a simple example of model checking with UPPAAL is discussed. The train gate controls access to a bridge for several trains, as a railway control system. The bridge is a critical shared resource which can be accessed by only one train at one time. The system is defined as four trains and a controller. A train can not be stopped instantly and restarting also takes time. Therefore, there are timing constraints on the trains before

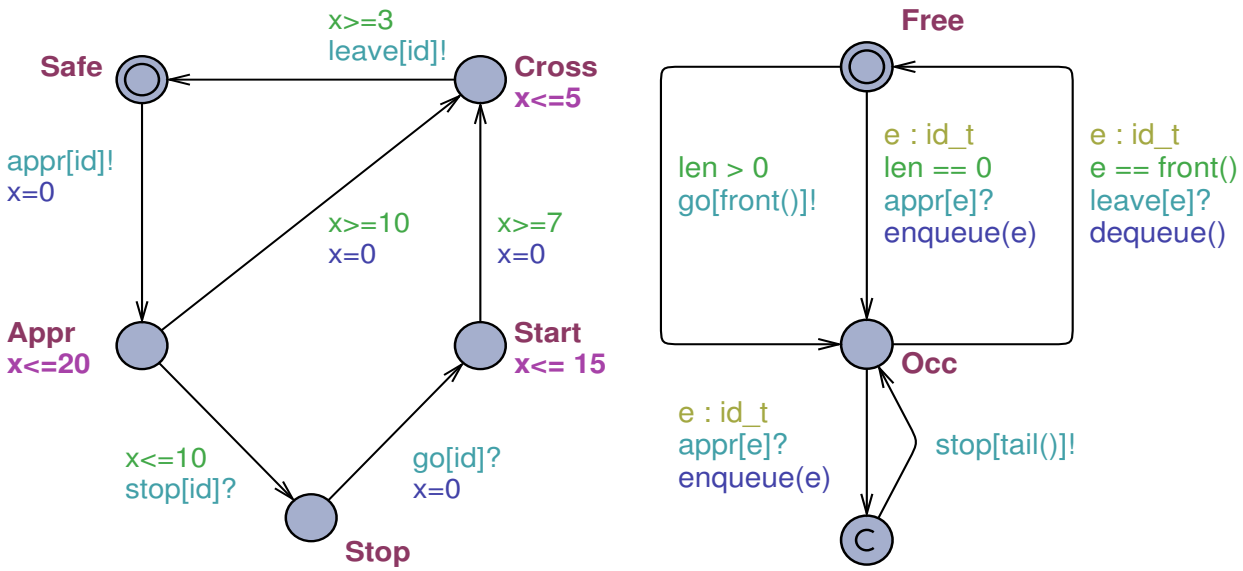


Figure 2.3: (a) The train model and (b) the gate model in UPPAAL

entering the bridge. When a train approaches, it sends an *appr!* signal. Thereafter, it has 10 time units to receive a stop signal. This allows the train to stop safely before the getting to the bridge. After these 10 time units, it takes further 10 time units to reach the bridge if the train is not stopped. If a train is stopped, it resumes the course when the controller sends a *go!* signal to it after a previous train has left the bridge and sent a *leave!* signal. After 7 time units, this train passes through the bridge and send a leave! signal after 3 time units.

The model of the train gate has two templates. Train is the model of a train, and Gate is the model of the gate controller and the queue of the controller, shown in Fig. 2.3.

We check the simple properties and deadlock. The simple reachability properties are expressed as:

- E <> Gate.Occ : Gate can receive (and store in queue) msg's from approaching trains.
- E <> Train(0).Cross : Train 0 can reach crossing.
- E <> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop) : Train 0 can cross bridge while the other trains are waiting to cross.

The following safety properties hold.

- A[] forall (i : id_t) forall (j : id_t) Train(i).Cross && Train(j).Cross imply i == j : There is never more than one train crossing the bridge (at any time instance).
- A[] Gate.list[N] == 0 : There can never be N elements in the queue (thus the array will not overflow).

The form of the liveness properties is Train(0).Appr→Train(0).Cross. Whenever a train approaches the bridge, it will eventually cross. To ensure the system is deadlock-free, we verify the query A[] not deadlock.

Through model checking of the train gate system, we show what may happen. A train may cross the bridge but the following trains will have to stop.

Chapter 3

Construction of FlexRay Model

3.1 Overview of FlexRay Communication Protocol Specification

This section gives explanations of the core of FlexRay communication protocol specification which are essential in building FlexRay model in UPPAAL. FlexRay protocols are first designed using specification and description language (SDL). For a automotive system based FlexRay, several nodes are connected by using star, bus and other topologies, forming a functional system, which is called a cluster. Each node has bus driver to connect to bus. A node consists of an ECU, a controller host interface (CHI) and a communication controller (CC). Fig. 3.1 shows the structure of node in the FlexRay bus.

Applications of communication system run on ECU of each node, which is a real time kernel that contains two schedulers for static cyclic scheduling (SCS) and fixed priority scheduling (FPS). ECU and CC share control and configuration information in CC, and also the processed data of ECU are stored in it. The CHI processes the data and controls the flow between the ECU and CC within each node. The CHI has two major interface blocks, the protocol data interface and the message data interface. The protocol data interface deals with all data exchange with regard to the protocol operation, while the message data interface manages all data exchange with regard to the exchange of messages as illustrated. The protocol data interface manages the protocol configuration data, the protocol data and the protocol status data. The message data interface handles the message buffers, the message buffer configuration and the message buffer status data.

The CC carries out the FlexRay protocol services. CC is the vital part in FlexRay communication model. The CC structure is defined in the protocol specification of FlexRay communication system, as shown in Fig. 3.2. It is divided to six parts as depicted as follows. Protocol Operation Control (POC) monitors overall status of CC and manages other parts. Media Access Control(MAC) is based on a recurring communication cycle. During one communication cycle, FlexRay provides the choice of two media access schemes, TDMA and flexible-TDMA. Frame and Symbol Processing (FSP) checks the correct timing of frames and symbols concerning the TDMA scheme, applies further syntactical tests to received frames, and checks the semantic correctness of received frames.

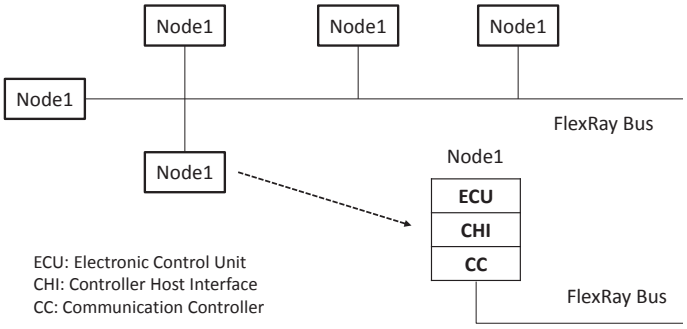


Figure 3.1: The topology of FlexRay bus system

Bus driver deals with coding and decoding processes. Macrotick Generation (MTG) controls the cycle and macrotick counters and applies the rate and offset correction values. The clock synchronization processing (CSP) performs the initialization at cycle start, the measurement and storage of deviation values, and the calculation of the offset and the rate correction values. The clock synchronization startup processing executes the initialization and commencement of the MTG process and the CSP process. This process has the repetitive tasks of measurement and storage of deviation values and the calculation of the offset and the rate correction values. Bus driver directs coding and decoding processes.

3.2 Abstracted FleRay Model

We abstract FlexRay communication protocol specification from the following three aspects.

Firstly, we focus on the communication between nodes and assume that the communication is in the case of clock synchronization and channel is no noise. The verification of design model of applications only needs functionalities of sending and receiving frames to be present in the FlexRay model. Therefore, only essential and necessary components for providing communication functionalities are selected. So the clock synchronization processing, the macrotick generation and the clock synchronization startup processing are out of our scope in CC. Instead, we use a simple mechanism for synchronization and startup between nodes. Moreover, the coding and decoding behaviors belong to the physical layer channel. It has nothing to do with the communication clock. So the bus driver is removed.

Secondly, because the clock is synchronized and no interference in the normal communication case, many function modules of CC become unnecessary. it is not necessary the FlexRay model to have the full behavior in sending and receiving frames. Behavior such as adjustments for error control, i.e. fault tolerance feature, can be skipped. So each module function has also been simplified on the basis of the communication function, and the details will be discussed in the next section.

Thirdly, according to our practical experience, we further simplified the structure of

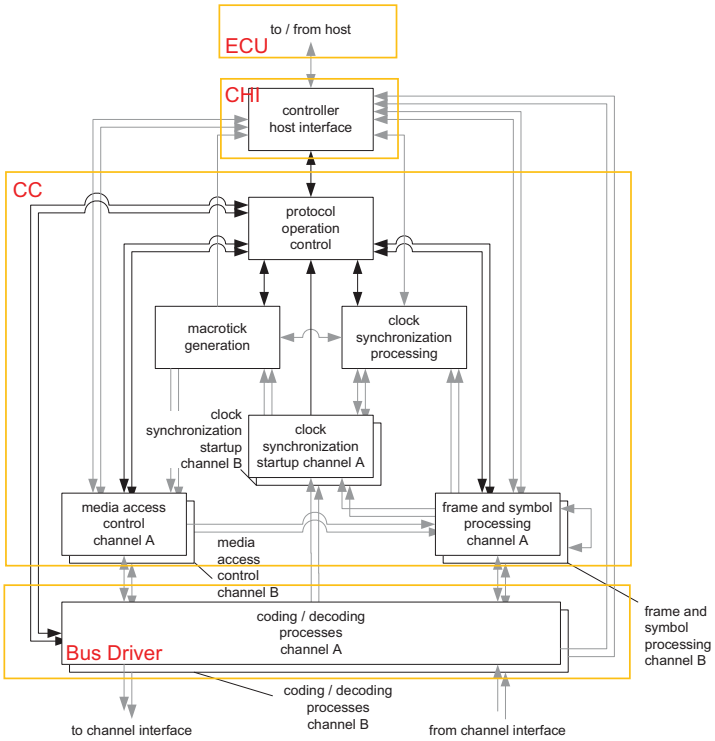


Figure 3.2: The structure of node

node. Due to the FlexRay system is very large and has a lot of states, out of memory happens when we check some properties in UPPAAL. Thus, we abstract FlexRay model again. A CC manages communication of all nodes. Due to FlexRay protocol is based on TDMA scheme, only one node can take channel In a moment. So we can through a CC to monitor all nodes, which send data in the specified time.

Fig. 3.3 is the simplified structure of nodes. In ECU, one or more tasks run on it. Different tasks can be designed according to the function of system in UPPAAL. In CHI, buffers are allocated for each ECU, where the generated data of the task is stored to the corresponding buffer. For CC, this part realizes the time driven function of the FlexRay protocol. For each part, we model their behaviors of sending and receiving messages, where these behaviors are related to time. POC monitors overall status of CC and manages other parts when system normally works. Timer controls the length of time slot. MAC is responsible for sending messages in special slots. FSP is responsible for receiving messages and transmiting data to tasks in ECU. By using the bus module as a buffer of storing frames, the transmission channel can be simulated.

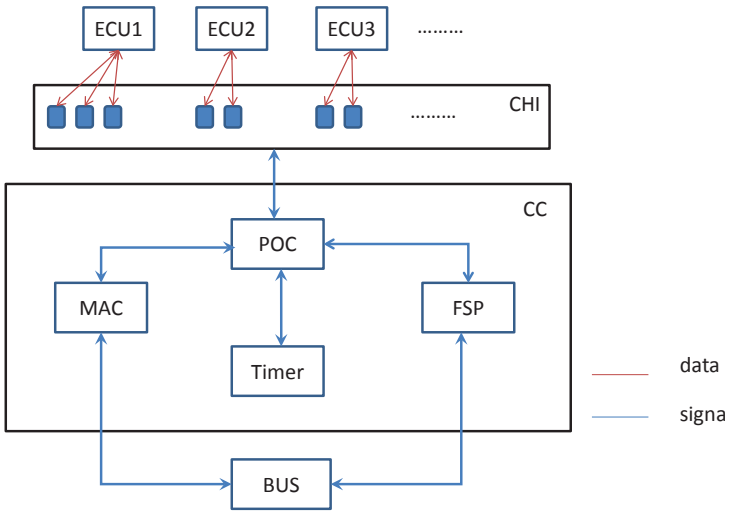


Figure 3.3: Simplified structure of node

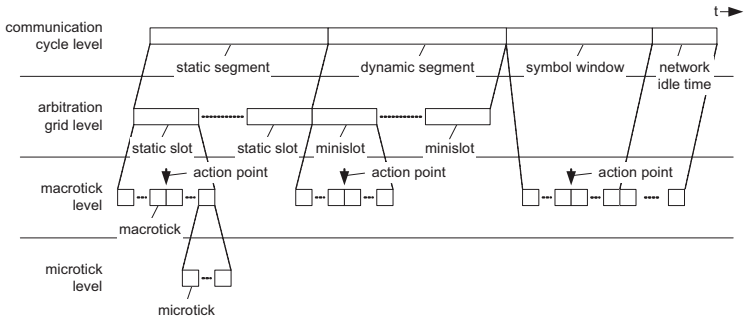


Figure 3.4: Timing hierarchy within the communication cycle

3.3 FlexRay Model

3.3.1 Communication Cycle

In FlexRay, communications take place in periodic cycles. In a cycle, FlexRay provides two kinds of communication schemes, the static TDMA and dynamic minislot-based FT-DMA with more flexibility. The time unit in the communication cycle can be divided into 4 parts: micritick level, macrotick level, arbitration grid level and communication level, as shown in Fig. 3.4. The length of microtick is determined by clocks of the FlexRay commination controller, where different nodes uses different clocks. The macrotick consists of several microticks, and the amount is determined by the clock synchronization mechanism. It should be noted that microtick is the minimum time unit for guaranteeing the overall clock synchronization. Multiple macroticks can compose one static slot in the arbitration grid level, or one minislot in the dynamic segment. There is an action

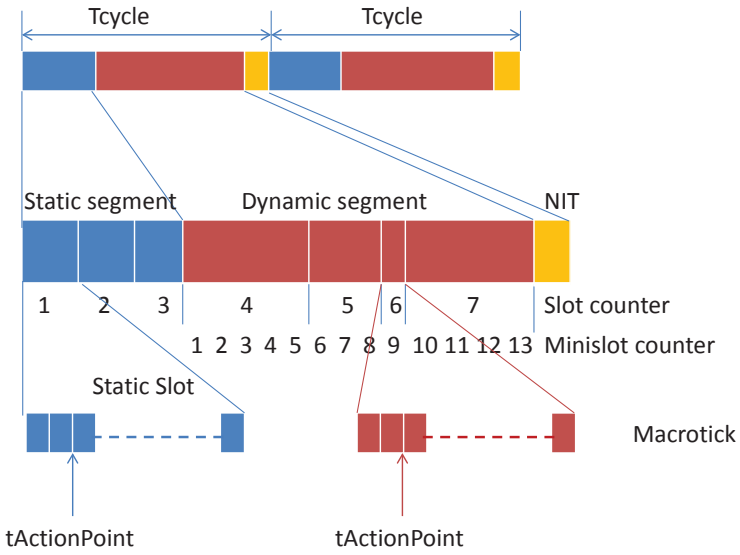


Figure 3.5: Communication cycle of FlexRay model

point in both the static slot and the minislot, as the offset from the its starting point. Arbitration grid level is the core part of FlexRay system, which determines the way of communications. In the highest layer a communication cycle contains the static segment, the dynamic segment, the symbol window and the network time (NIT). Both the static and dynamic segments are comprised of several corresponding slots, which are periodically repeated. The symbol window is a transmission period in which a symbol can be sent on the network. It is responsible for synchronizing the cycle between nodes. The symbol window contains a configurable number of maroticks. If no symbol window is required, it is possible to configure zero maroticks. The network idle time is a communication-free period, which contains all the macroticks which are not used in the previous three time units in one cycle. And it also reset the value of slot counter. The dynamic segment and the symbol window are not necessarily needed for every cycle, and the durations of them are adjustable as required. In our research, we ignore the symbol window, because in normal communications, it doesn't need treatment the synchromism symbol between nodes and do not influence the time of sending message. And we only consider the time from the communication cycle to the macrotick level, as shown in Fig. 3.5.

The static segment uses TDMA to realize time-triggered bus access, with fixed time duration for each cycle. Static slots are numbered in ascending order from the very beginning. One or more static slots be allocated to the nodes. The allocation method is unchangeable during the operating, and the allocated static slots can only transmit message for specific nodes. Even when one static slot does not contain information data, the system will keep idle until the next static slot comes.

The dynamic segment is event-triggered, based on the mini-slot TDMA, and it is comprised of a fixed amount of minislots. The time duration of the dynamic slot depends on the length of the message, which requires one or more minislots. The system will start the

next slot after other nodes finish receiving message.The dynamic slots also be allocated to different nodes which send the dynamic message only in specified dynamic slot. If there is no enough minislots for the message to transmit in one cycle, it will wait and check the next period. What if there is no information to send within a dynamic slot, all nodes will wait a minislot. Then dynamic slot counter will increase until the next minislot comes. The system will keep this procedure until the dynamic segment ends. If all the required transmissions finish before the dynamic segment ends, the system still has to wait for the rest of the segment duration.

According to the overall clock, different nodes determine the time for sending or receiving a specific data frame with a particular ID, which is called the slot ID. To determine this specific time, every node uses a slot counter for both the static and dynamic segments, with the counter value being the current slot ID. During every static slot and dynamic slot, only one node is allowed to send message on the bus. Therefore, at the beginning of every slot, all the nodes have to check whether the message ID and the slot ID match with each other. If and only if they are matched, this message can be transmitted.

For static messages, ECU uses static cyclic scheduling(SCS), and during the period of the system design, the slot identifiers are allocated to nodes. The static messages are send according to the schedule table in ECU and the length of the transmitting data should abide by the specification defined in *gPayloadLengthStatic*. For dynamic messages, ECU uses fixed priority scheduling(FPS), and the message with smaller ID will be sent with higher priority. In one communication cycle, if there are a large number of nodes transmitting information, or the message length is too long, there is no enough minislots for the message with large ID to be transmitted in the period. In this case, it will wait for the next cycle. Obviously, it is quite uncertain that all messages are able to get the required bandwidth, which depends on how much messages has been transmitted previously. Hence, for the information data with strict delay requirement, it is essential to transmit it in the static segment, or allocate smaller ID (higher priority) in the dynamic segment.

There is an example of FlexRay communication cycle, as shown in Fig. 3.6. This figure shows 3 nodes and how the messages are sent. At the beginning of the communications, CC configures the relevant parameters, with the slot counter and minislot counter returning to 0. There are the first two communication cycles, which are used to demonstrate the timing of message transmissions. Each node has two buffer queues to store static and dynamic messages in CHI. For the ST message, every ECU has a scheduling table of the transmission time. When it is the time of sending ST message, ECU will store the operation results into the buffer with relevant ID. Every static message is assigned with the cycle and slot numbers for transmitting in the specific time. For instance, if the message 'Ma' is assigned to the first cycle and the second slot by 2/1, it will be sent during the second slot of the first cycle. Similarly, 'Mb' will be transmitted in the first slot and of the first cycle and 'Mc' will be sent during the first slot of the second cycle .

For dynamic messages, every dynamic message is assigned with a slot ID. The dynamic messages are packed into frames by the bus driver with a frame ID being identical to the allocated slot ID. This ID also represents the priority in sending messages. For instance,

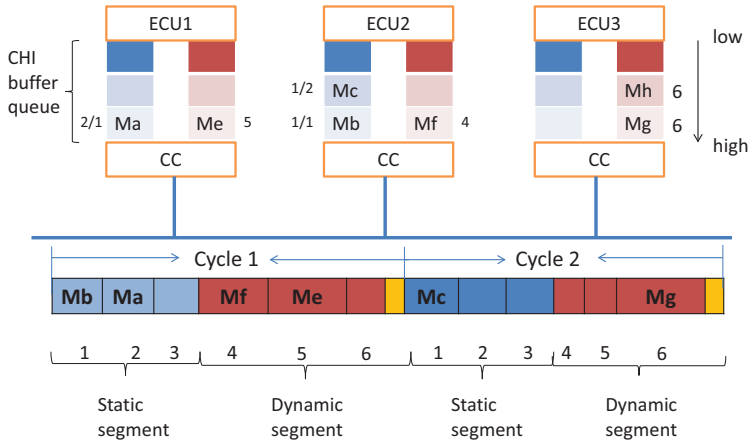


Figure 3.6: FlexRay communication cycle example

the message 'Mf' has ID 4, so it is the forth slot to be sent in the first cycle. Then, the message 'Me' with ID 5 is sent in slot 5. Besides, if there are more than one messages shared the identical ID, like 'Mg' and 'Mh', to be transmitted within a same cycle, ECU will determine which one will be first transmitted according to the priority scheme. The messages with higher priorities will be sent in the current cycle. Clearly, 'Mg' has higher priority than 'Mh'. But, the dynamic segment in first cycle does not has enough minislots for it. So 'Mg' is sent in slot 6 of the second cycle. If there is no message to send in the slot, like slot 6 in the first cycle, this slot still consumes a minislot.

As can be seen, the time of static messages transmission is predictable and its access to the bus is deterministic. However, by assigning time slots for nodes and messages to fix the bandwidth allocation, the bus bandwidth may not be fully used, and it is not flexible for the later nodes expanding. Due to these reasons, dynamic segment makes up for the weakness. It guarantees that messages with high priority still have the chance to be transmitted even when the bus is busy. The messages with low priorities will be sent when the bus is idle. In this case, nodes can share the bandwidth and the bandwidth can be dynamically allocated with more flexibility. The determinacy of the bus access can be guaranteed and the drawbacks of the static transmission are compensated.

3.3.2 Protocol Operation Control

In this sub-section, we discuss how CC is defined in different parts of the FlexRay communication protocol specification, as well as the structure realization of the FlexRay model in UPPAAL. Every node of the FlexRay system has 6 basic Protocol Operation Control States and different functions are realized in different states, respectively. Hence, when the node ECU starts, the node states have to be transformed according to the relevant protocols for the normal communication. The stares transition diagram is shown as flows Fig. 3.7.

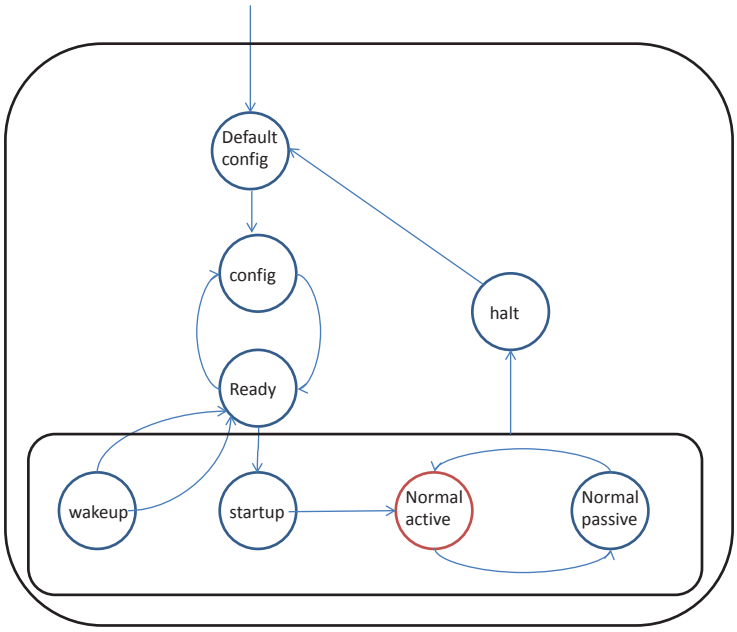


Figure 3.7: Overview of protocol operation control

- *Config* status (including *Default Config* and *Config*): for initializations, including the communication period and data rate;
- *Ready* status: for the internal communication set;
- *Wakeup* status: to wake up the nodes which are not communicating. In this sate, one node sends the wake-up signals other nodes, to awaken and activate the communication controller, the bus driver and the bus monitor;
- *Startup* status: to start the clock synchronization, as a preparation of the communication;
- *Normal* status (*Normal Active* and *Normal Passive*): normal active communication status;
- *Halt* status: to indicate that the communication is in outage.

The FlexRay system has to be first initialized before getting to the ready status. Then, the node states and the relevant invariants are updated according to the CHI. During certain time duration, some nodes or the communication cluster do not work and are in the power save mode. They should be awakened before being back to work, and the awakened nodes can also wake up other nodes and the whole cluster. Next, the synchronous frames are sent during the same slot of every cycle. The slot counter and the cycle counter are initialized between nodes, and the clocks are synchronized by the clock synchronization startup module. At this moment, the POC starts MAC and PSF modules and then the system can achieve the normal communication status. In the communicating process, when POC receives the preempting deferred commands from other parts of the CC, and then it will jump to the halt state. Before the next communication cycle start,

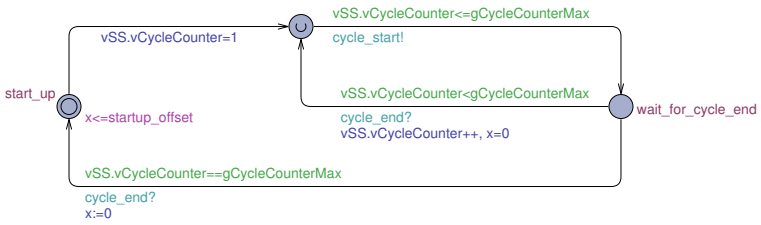


Figure 3.8: Protocol operation control model in UPPAAL

if the error checking sends signals, POC will change the state according to the signals. For instance, if the node is configured not to allow the communication to be halted due to severe clock calculation errors, than POC will transfer to normal passive status.

This research focuses on the properties of the communication time, which ignores the processes of the system launching and synchronizing. We only emphasize the data transmission and response in the normal communication status, and perform verifications to them. Therefore, only the model of POC in the normal active status is provided in this research. The diagram in UPPAAL is given by Fig. 3.8.

In the normal active status, the system has already finished initialization and synchronization procedures, and POC starts to execute a sequence of tasks at the end of each communication cycle. We use the *start_up* as initial state. In this state, the time constrained by the invariant *x<=startup_offset* is smaller than that by *startup_offset*, before the transmission. *startup_offset* is a constant defined as 0, which indicates that the system starts timing when the normal communication begins. Before the first transmission, we set the *Vss.vCycleCounter* equal to 1, starting the first cycle. The POC model will check that if the current cycle number is less than or equal to the max cycle value *gCycleCounterMax*, it will send a *cycle_start!* signal to MAC, and MAC will start the first slot of static segment. Then POC will go to the *wait_for_cycle_end* state and wait until this cycle ends. At this moment, the first communication cycle begins, including static segment and dynamic segment. When the last minislot in the dynamic segment finishes, MAC will send a *cycle_end!* signal to finish the first communication cycle. POC will again check which cycle it is at this moment. If *Vss.vCycleCounter* is less than *gCycleCounterMax*, *Vss.vCycleCounter* will plus one and the next cycle starts. if *Vss.vCycleCounter* is equal to *gCycleCounterMax*, POC will reset the system timing, re-configure the system and redo the previous processes.

3.3.3 Media Access Control

The media access control (MAC) deals with the usage of the public channel for the communication clusters when there is a competition. It is the most important part of the FlexRay model. In the FlexRay protocol, MAC is based on a recurring communication cycle, and it control the segments allocation during the communication cycle. Different media access schemes are employed in static segment and the dynamic segment. One cycle duration is divided into many slots with their labels in order, and the slots with

different ID are allocated to the relevant nodes. Therefore, frames sent from one node have the same ID at the head of them. The frame should only be sent during the time slots with the same ID, which efficiently avoids the nodes occupying the channel. The execution of MAC in detail is expressed by SDL in the FlexRay protocol specification, as shown in Fig. 3.9.

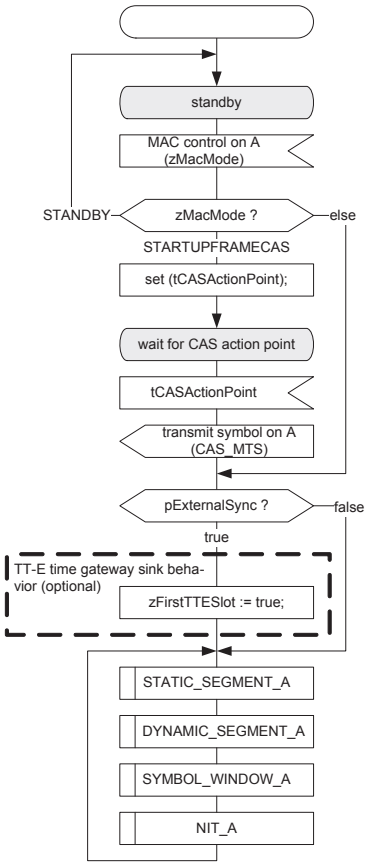


Figure 3.9: Media access process

After the system starts, MAC will receive *STANDBY* command sent by POC, wait at the ready position and make decisions based on the received *zMacMode*. If it is *STANDY*, MAC will reset the internal communication and be back to the ready position; if it is *STARTUPFRAMECAS*, it indicates that the collision avoidance action point need to be reset, and *CAS_MTS* testing symbol should be sent for CSMA/CD during this period. The MAC checks whether this node is synchronized with the other external nodes. If the value of *zMacMode* is none of the two mentioned above, the system will jump over the collision avoidance operation and directly check whether this node is synchronized. If the *pExternalSync* value is true, *zFirstTTESlot* has to be set true, which belongs to the TT-E time gateway sink behavior; if the *pExternalSync* value is false, the system will jump over this step and repeatedly start the communication cycle from static segment.

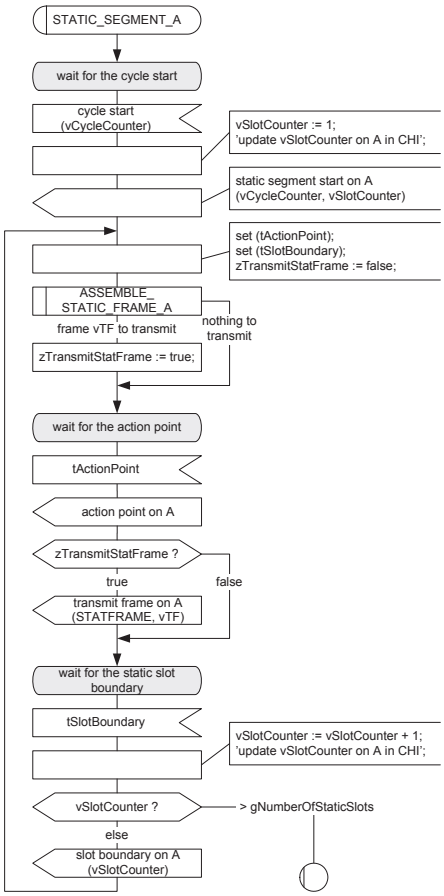


Figure 3.10: Media access in static segment

The MAC for static segment is shown in Fig. 3.10, and the main functions are slot number counting, action point setting, frame assembling, frame transmitting and process cycling\starting dynamic segment. First of all, the system wait for the cycle start until the cycle counter start counting. The initial value of the slot counter is set at 1 by MAC when starting the first static slot, and the *vSlotCounter* value in CHI has to be updated. Then, the channel is informed that the static segment can start and the values of current cycle and slot can be assigned. The action point and slot boundary time of the static slot should be set and let *zTransmitStatFrame* be false. If there is no information data to be sent, MAC jump back to wait for the action point state. When the action point time is out, the channel is informed, and MAC will check *zTransmitStatFrame* and see whether there are information data to be sent. Specifically, if it is true, frame transmission takes place. Otherwise, MAC will go to *wait_for_the_static_slot_boundary* state until this static slot ends. At the time edge of this slot, the slot counter will plus 1, and the slot counter value in CHI will be updated. Finally, the slot counter value is checked. If it is no larger than the maximum value of the static slot number, the next static slot will start and the parameters such as the action point will be reset; otherwise, it indicates that the static

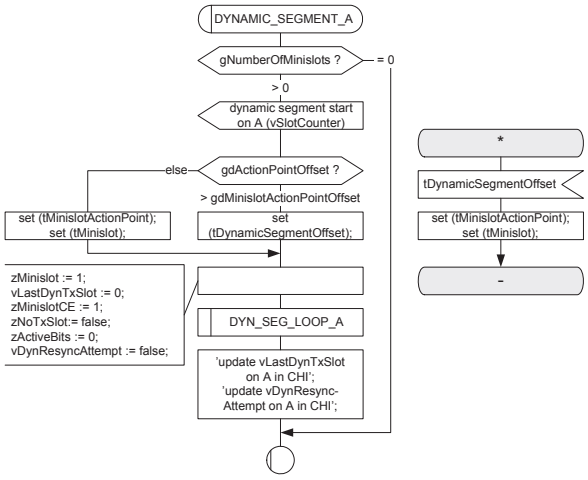


Figure 3.11: Media access in dynamic segment

segment ends and dynamic segment should start. The SDL diagram of dynamic segment is shown as follows Fig. 3.11.

The process of the dynamic segment is similar with that of the static segment, however, with higher complexity. The MAC will first check whether minislots are assigned for a communication cycle when starting the dynamic segment. If the minislot number (*gNumberOfMinislots*) is 0, it indicates that there is no dynamic segment in the system; if it is larger than 0, the dynamic segment will start. Before the dynamic segment cycling, the system has to compare the offset time durations of the static slot and the minislot. If the offset time of the minislot is longer than that of the static slot, the *actionpoint* should be set at the first minislot of that dynamic slot. Otherwise, a dynamic segment offset is added and this time duration plus the minislot offset equals to the static slot offset. It has to be noted that this happens only at the first action point of the dynamic segment, and in other cases the minislot action point is used as the offset. This process is shown in Fig. 5.2 as the timing at the boundary between the static and dynamic segments. Then, the local information has to be updated, such as the current minislot number, the dynamic slot counter value, the communication elements in minislots and the detection of a possible slot counter desynchronization (*zNoTxSlot*). At the end of the dynamic segment loop, the value of the last dynamic transmit slot counter is updated and the dynamic segment should be re-synchronized. Then the NIT segment start directly.

The dynamic segment loop is briefly introduced according to the SDL Fig. 3.13 of the FlexRay protocol specification. When every dynamic slot starts, the node will check whether there is enough time left in the dynamic segment for transmission, or whether there is no transmission allowed in this dynamic slot due to the detection of a possible slot counter desynchronization as indicated by the variable *zNoTxSlot*. MAC will perform the assembling dynamic frame process if a transmission is allowed. If the frame is empty, it indicates that there is nothing to transmit; if the frame is not empty and also need to be transmitted in the current dynamic slot, then it will be sent. If the node does not

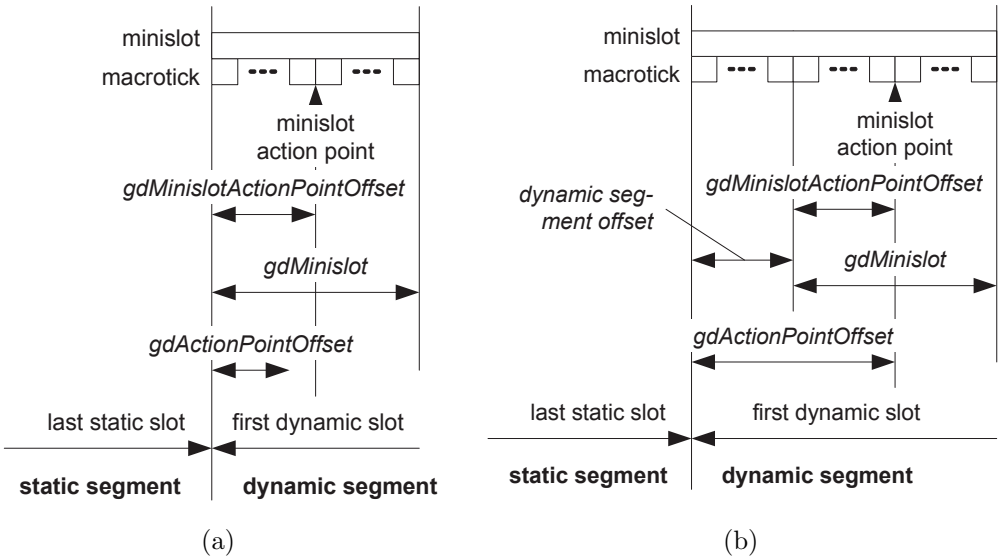


Figure 3.12: (*a*) action point a (*b*) action point b

transmit itself, it will wait the transmissions of other nodes in the MAC until the end of the minislot state. In the case that there is no *CE start* signal detected before the minislot ends, the node jumps to the next dynamic slot process. If a *CE start* signal is detected, the node notes the current minislot and update data of the channel, which indicates that there is one node starting to send frame during this slot. MAC waits for the end of activity.

If DTS is detected which is indicated by the CODEC process, a fault-free frame reception will also enable the detection of the DTS. As soon as the DTS was received, the node locks down the end of the dynamic slot, with the intent that potential noise during the succeeding idle detection cannot affect the remaining dynamic slot length. The last potential idle start signal before the *CHIRP* signal marks the minislot in which the frame transmission ended, and is used to derive the last minislot of the dynamic slot. Should the communication element end before the number of bits crosses the *cFrameThreshold*, the communication element is regarded as noise and the node tries to switch to a state where no noise was received. It does so by not applying the *gdDynamicSlotIdlePhase* lengthening of the dynamic slot on the one hand and by increasing the dynamic slot counter by two should a minislot boundary have occurred between the *CE start* signal and the *CHIRP* signal.

After the reception of the *CHIRP* signal, the node awaits the end of the dynamic slot. A CE start signal at this point in time is generally an indication of a fault on the bus; either the preceding or the current communication element was noise or a frame transmitted due to a fault condition. In case that the preceding element was already categorized as noise due to its short length, the node treats the new communication element as frame and potentially adjusts the dynamic slot counter. Under normal circumstances, no CE start signal will be received during the*wait for the end of the dynamic slot* state and the

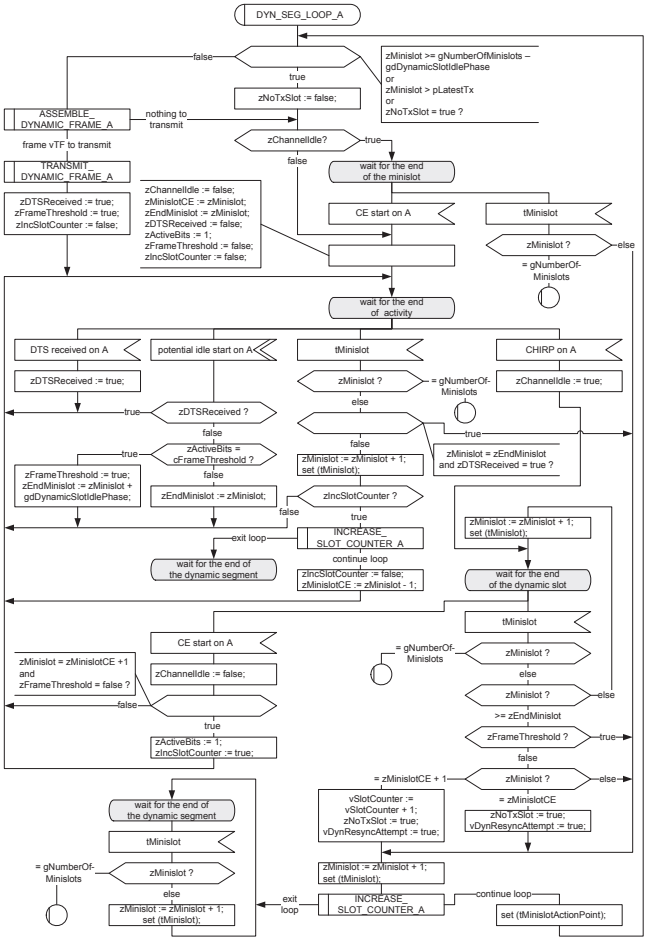


Figure 3.13: Media access in dynamic segment arbitration

dynamic slot will end at the end of the minislot where *zMinislot* is equal to *zEndMinislot*. The end of the dynamic slot causes the dynamic slot counter to be incremented and then exported to the CHI. If the received communication element was shorter than the frame threshold *cFrameThreshold* and the dynamic slot was either one or two minislots long the node will abstain from transmitting in the following dynamic slot and a resynchronization attempt is noted for indication to the CHI at the end of the dynamic segment. If the received communication element was shorter than the frame threshold and the dynamic slot was two minislots long the dynamic slot counter is incremented twice instead of just once, as is normally the case.

When the system is synchronized and at the normal communication status, MAC will omit the collision avoidance operation and the clock synchronization process. Hence, the MAC model in our research directly starts the communication loop and ignore the symbol window. The MAC model in the static segment is almost in accord with the specification. The only difference is that we do not provide the assembly process of the frame, while instead, we use a simple data structure to substitute the frame format. Due to the

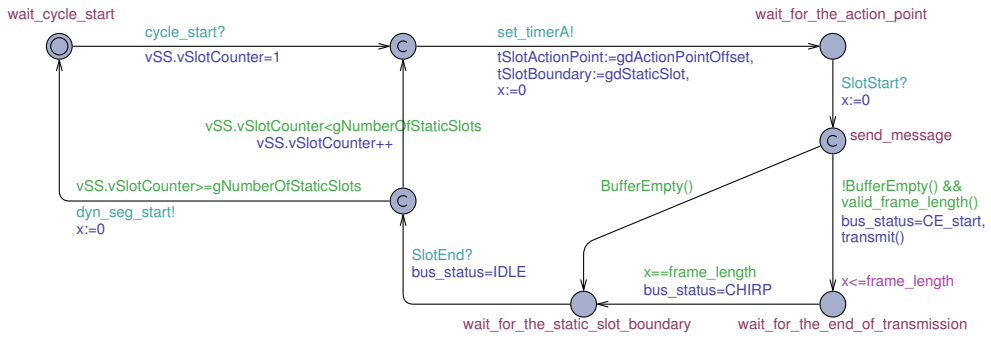


Figure 3.14: MAC static segment model in UPPAAL

complexity of the dynamic segment, we only extract the key processes of the with regard to the minislot and dynamic slot, without considering the bit streams and noise problems (the channel is assumed to be no-interference). In addition, if a frame occupies more than one minislot, we take them as a whole. When the frame ends, MAC will compute which monislot the frame is, according to the data length. It dose not influence the calculation of the transmission time and response time. In our model, *tslotend* is used to specify the ending ponit in different time duration. such as actionpoint, dynamic segment action point and every dynamic slot ending point. Set *timerB!* is for time setting. In the following part, we will introduce the static and dynamic segment models in UPPAAL.

The media access processing in static segment is shown in Fig. 3.14.After receiving signals of a cycle from POC, MAC starts from the first static slot and *vSlotCounter* is equal to 1. In each static slot, MAC sets timer, sends signals to MGP process and set the length of the action point in the static slot (the boundary of the static slot). Then, MAC waits signals of timings for sending a message at *wait_for_the_action_point* state. When *SlotStart* is received, MAC decides whether to send a message or not and waits for *SlotEnd* signal. If buffers in CHI with certain ID are empty, which means that there is no data to send, MAC will wait until the current slot ends. If there is a requirement for sending a message and the length of this massage is a valid number(it can be sent out during this slot), transmission starts and MAC sets bus status to *CE_start*. MAC changes the bus state to *CE_start* and starts transmitting. Then it wait at *wait_for_the_end_of_transmission state* until the transmission ends. When the clock x satisfies *x==frame_length*, the bus state will change to CHIRP and wait for the current slot ends. After receiving the *SlotEnd* signal, the bus turns to idle. If the slot counter is less than the defined maximum number, MAC increases the slot counter and jumps to next slot. When slot counter reaches the maximum value, the static segment ends. MAC sends a *dyn_seg_start!* signal to dynamic process and returns to the initial state.

When the static segment ends, dynamic segment starts, as shown in Fig. 3.15. Due to the complexity of the dynamic segment, we mainly focus on the key steps of computing the minislot and dynamic slot, without considering channel bittorrent and noise problems. In addition, if a frame contains continuous multiple minislots, they are considered as a whole. At the end of the frame transmission, MAC calculates which minislot

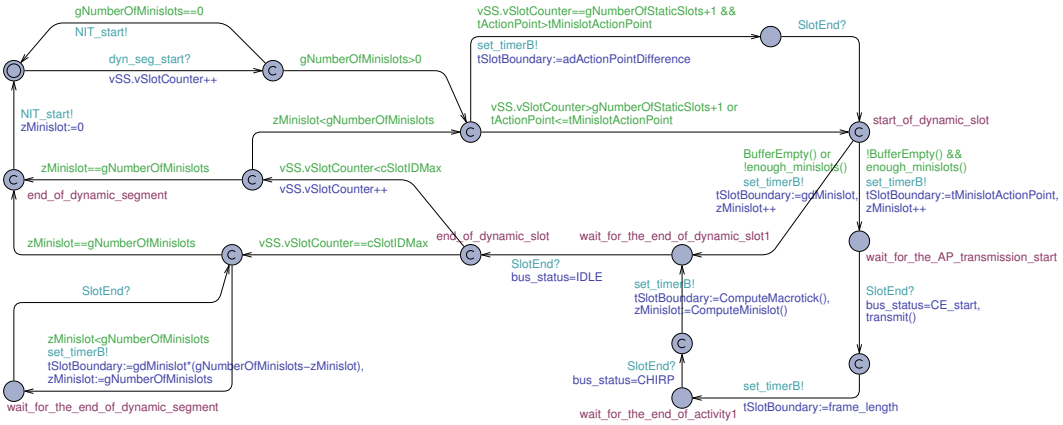


Figure 3.15: MAC dynamic segment model in UPPAAL

the frame is in, according to frame length. This does not affect the time calculation of transmission and response. In this work, we use the Timer to realize it. Firstly, MAC checks if *gNumberOfMinislots* is larger than 0. If it is, there is no dynamic segment in this system. Then this communication cycle ends and MAC sends the *cycle_end!* signal to POC. If the answer is yes, MAC starts dynamic segment and slot counter++. Then MAC sets timers of action point and ends the minislot to start a dynamic slot. If it is the first slot in this dynamic segemnt, and static actionpoint is larger than minislot actionpoint the gap between then is dynamic segment offset. In this period of time, we use the Timer for calculation. When this time duration ends, it is equivalent to the case when static actionpoint is no larger then the minislot actionpoint, and MAC also stays in *start_of_dynamic_slot* state. In a dynamic slot, MAC checks if the current slot ID is specified for transmission (whether there are data ready in CHI buffers matching the current slot ID) and if there are enough minislots for transmission. If there is no information to be sent, or there are no enough minislots for data transmission, when the current minislot ends, this dynamic slot also ends and minislot plus 1. Also, in the case of no transmission, if a transmission by another node is detected, we have to wait until CHIRP is detected. In another case, the specified buffer is not empty and there are enough minislot in dynamic segment for transmission. MAC will set the actionpoint of minislot. Then, MAC stays in *wait_for_the_AP_transmission_start* state and wait for transmission. When actionpoint comes, the bus state changes to CE_start, the information data in CHI buffer will be sent. According to the information length, MAC determine the length of the dynamic length. MAC stays in *wait_for_the_end_of_activity* state and wait until the frame transmission ends. When the frame transmission ends, the bus state changes to CHIRP at the channel idle recognition point. Due to the fact that a transmission can use several minislots, we have to count minislots between *CE_start* and *CHIRP*. MAC updates current values of the minislot and the macrotick. Then we use the return value of *ComputeMacrotick()*, which indicating the number of macrosticks left in the minislot when the frame transmission ends, to set the timer.

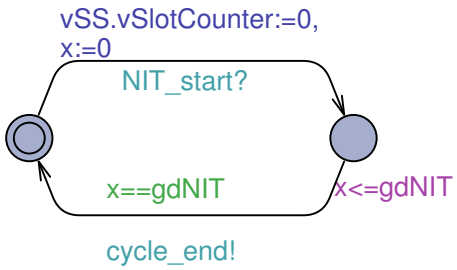


Figure 3.16: Network idle time model in UPPAAL

As shown in Fig. 3.15, after receiving the dynamic slot ending signal, the bus state changes to IDLE. MAC checks that whether the slotcounter and the minislot reach the maximum values. If both of them do not, slot counter increase, the next dynamic slot starts. If the slot counter is less than the maximum value but minislot value is the maximum, or both the slot counter and minislot values reach the maximum, then the dynamic segment ends and we make *zminislot* equal to 0 and wait until the next cycle. If the slot counter value reaches the maximum, while minislot value is less than its maximum values, no dynamic slot is possible in the segment, we should count the remaining minislots utill *gNumberOfMinislots* is reached. Then the dynamic segment ends and therefore a communication cycle finishes.

After the end of dynamic segment, there is NIT of each communication cycle. It contains all the macroticks which are not used in the previous static and dynamic segment. Fig. 3.16 shows the processing of NIT in UPPAAL. NIT resets slot counter value when receiving *NIT_start* signal. While NIT ends, the current communication cycle end and turn into the next cycle.

3.3.4 Timmer

The reason why the arrival time of the information data can be known in advance in the FlexRray protocol is that all nodes in the communication clusters follow a standard global, which guarantees the determinacy of the system. However, it does not follow that all nodes exactly keep the same synchronizing time. The truth is that the time differences of nodes are tolerated within a small error range (usually 1-2us). Macrotick is the shortest time unit which ensures the global time synchronization, which means that all macrosticks in nodes of the communication cluster should be defined by the same time duration.

In CC, the clock synchronization startup process executes the initialization and start the MTG and CSP process. The clock synchronization consists of two main concurrent processes. The first one is the macrotick generation process which controls the cycle and macrotick counters and applies the rate and offset correction values. It is realized by using the correction terms to adjust the number of microticks in each macrotick. The other one is the clock synchronization process (CSP), which performs the initialization when a cycle starts, such as measurement and storage of deviation values and the calculation of the

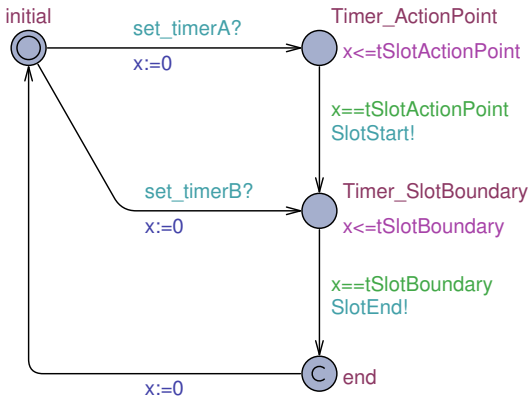


Figure 3.17: Timer model in UPPAAL

offset and the rate correction values. MTG, CSP and CSS jointly complete the processes of synchronization and modification which are inseparable. Due to the assumption that the synchronous transmission is error-free, this thesis mainly focuses on the research of channel allocation and the response time for nodes, without considering MTG, CSP and CSS. However, to ensure the correctness and conformity of every time unit, we use timer, instead of MTG, CSP and CSS, to calculate the time length of every slot and actionpoint within a cycle. The Timer model in UPPAAL is shown in Fig. 3.17.

Timer is used in both the static and dynamic segments. In the static segment, it controls the actionpoint and the ending time of every static slot. The operation process of Timer in static segment can be described as follows. Before receiving *set_timerA* from MAC_static model, Timer waits at the initial state. When detecting *set_timerA*, the clock will be reset and start timing in *Timer_ActionPoint* state. When clock x equals to *tSlotActionPoint* (static slot action point) which is set by MAC, Timer sends *SlotStart!* and informs MAC that the time requirement has been satisfied and transmission can start. Timer continues to count the macrotick and when the clock time equals to the length of the static slot, this slot ends. Then, the clock will return to 0 and wait for the next static slot.

For the dynamic segment, Timer not only calculates the action point but also controls two special time points: one is the time difference between static slot action point and minislot action point and the other one is the information length. Moreover, due to that the information length is expressed by macrotick, when the dynamic segment finishes sending message, it is needed to calculate the transmission ends in which minislot. Then, Timer will wait the rest of macroticks in this minislot and the whole dynamic slot will not end until this minislot finishes. In this model, we respectively use *set_timerB*, *tSlot-Boundary* and *SlotEnd* to represent the beginning time, different time durations and the ending time.

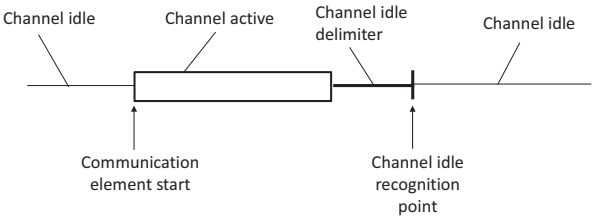


Figure 3.18: Reception related events

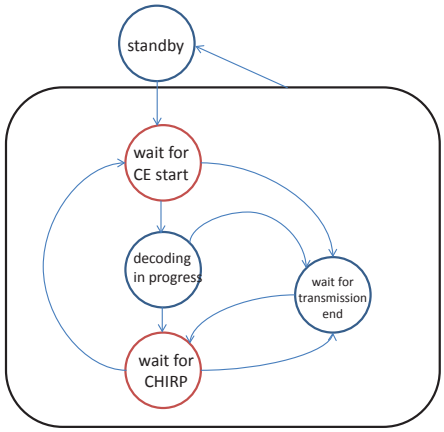


Figure 3.19: Overview of frame and symbol processing

3.3.5 Frame and Symbol Processing

Frame and Symbol Processing (FSP) is for the receiving data processing in both the static and dynamic slots. It checks the correct timing of frame and symbols with respect to the TDMA scheme, applies further syntactical tests to received frames, and checks the semantic correctness of received frames. Reception-related events for FSP are shown in Fig. 3.18. In the channel, when a frame is transmitting, the channel changes from idle to the active state. When the transmission ends, the channel idle delimiter confirms whether the channel is free. The channel idle recognition point marks the end of the slot. Then FSP waits for a certain time duration , until the end of the current slot or minislot. Fig. 3.19 gives an overview of the FSP related state diagram.

When CE starts, the communication element also starts. CHIRP denotes that channel idle recognition point is detected, which represents the frame has been completely transmitted. The system will wait until the transmission ends, which also means the current static slot/dynamic slot ends. The receiving process should start from the beginning of CE to CHIRP. Frame decoding does not affect the receiving process. Therefore, we select two states of the FSP model: *wait_for_CE_start* and *wait_for_CHIRP*, as shown in Fig. 3.20

Here the transmission ending time is equal to the slot finishing time, and we put de-

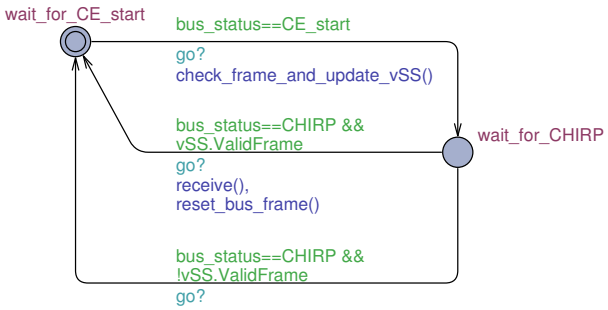


Figure 3.20: Frame and symbol processing model in UPPAAL

termine it in the MAC model and update the bus state to idle. When MAC starts to transmit frames, the bus will change from idle to *CE_start* state, and FSP can receive frames and check whether the ID and configuration are matched with each other. If they are matched, this frame will be labelled *Vss.ValidFrame* (valid frame), which is realized by *check_frame_and_update_vSS()*. When finishing the frame transmission and the bus state changes to *CHIRP*, FSP will store the data into the corresponding buffers, and empty the bus. If this frame is not valid, FSP will return to the initial state and wait for the next transmission.

Chapter 4

Model Checking Applications with FlexRay Model

In last chapter, we have been introducing how to abstract and establish FlexRay model in UPPAAL. FlexRay model is the core of our framework. In this chapter, we will describe how to build application model based on FlexRay model, how to connect application design model to FlexRay model by using interface and how to check communication systems design model which is composed of FlexRay model and application model. Then, we also show an example to explain them.

4.1 Model Checking Applications with FlexRay Model Process

In this section, the process of verifying automotive systems with the FlexRay protocol is presented. The flow chart is shown in Fig. 4.1. First of all, in accordance with the FlexRay communication system protocol specification, we abstract it and capture the core behaviors of the communication controller specified in FlexRay communication protocol specifications, and build a reusable FlexRay model. This part has been described in the previous chapter. Then, according to our framework, application model has been set up based on FlexRay model. Application model models the behavior of tasks in the ECUs, the main focusing point is the message transmission in the FlexRay model. While we do not consider the schedulers and just use the simplest model with timing of output and input. After that, we need connect the application model to FlexRay model, forming the communication system.

In order to more convenient check communication system with FlexRay, the FlexRay model also provides parameters and interfaces for communication for the application model to access. The application model links with the FlexRay model by setting the configuration. This part only needs to follow these parameters and interfaces to cope with the FlexRay model as an automotive system design model. There is a clear boundary between FlexRay model and application model in UPPAAL. In this sense, users can more simply design the application model, and do not needed to take into account of the

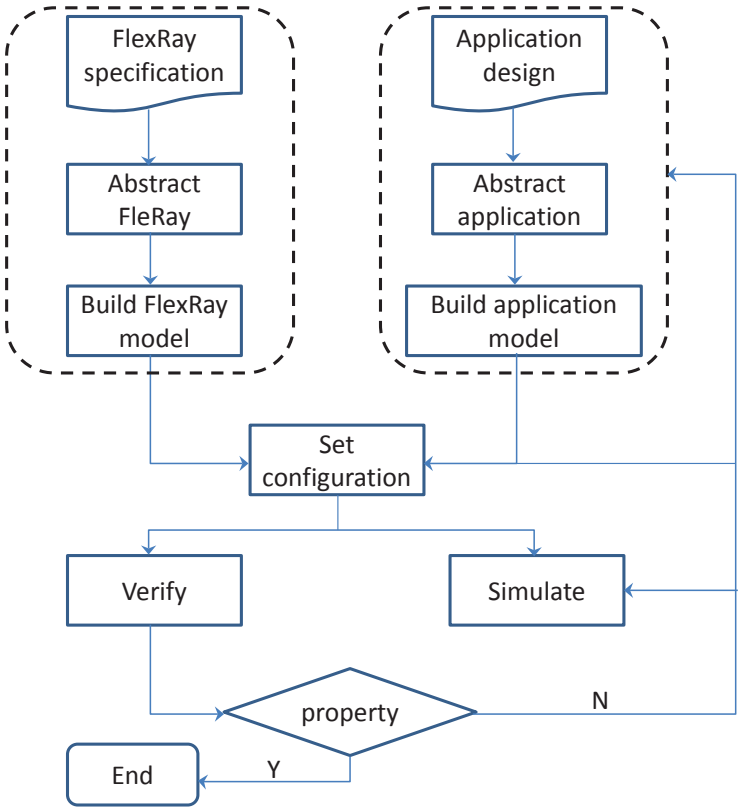


Figure 4.1: Flow chart of the checking model

specification of the FlexRay communication system protocol.

Finally, we implement the simulations and verifications by writing some queries in UPPAAL. we may to check properties of the component system design model, such as deadlock, deadline, response time and feasibility. If one property can not be fulfilled, we are able to find out bugs and revise the model or configuration again through the analysis of the trace.

4.2 Interface of FlexRay Model

In order to efficiently perform communication systems checking, FlexRay model provides an interface to connect application design model, which enables the ECUs of this application system satisfy the FlexRay communication protocols. The interface can be divided into two parts. The first one is the parameter settings of the communication cycle. The settings are given in the FlexRay communication protocol specification [18] and we select the relevant parameters relative to the FlexRay model. Those parameters mainly include the communication cycling time, the lengths of static and dynamic segments and the capacities of static slots and minislots, as described in Tab. 4.1. The second one is

configuration of application design model.

- We need define how many messages need to transmit in the communication cluster.
- The message are allocated in which slots of communication cycle sending.
  typedef int[*1, cSlotIDMax*] *t_msg_slot*;
  Type define for messages in nodes, [*1, cSlotIDMax*], denoting that the slot ID is used to identify each message. This means slots are allocated to different messages. Note, a slot can not be allocated to multiple nodes.
- We define a data struct to describe the CHI buffer, sending buffers and receiving buffers, which facilitate ECU's reading and writing to CHI. CHI works as the buffer storing the sending and receiving data, as well as the data length. The buffer ID and slot ID are corresponded to the message ID, therefore, message can only be stored into buffers and sent in slots, both with the same ID. The structure is defined as follows:
  typedef struct
  {
  int[0,*MaxDataValue*] *data*;
  int[0,*pPayloadLengthDynMax*] *length*;
  } Buffer;

  Buffer CHI_Buffer_send[*cSlotIDMax*+1];
  Buffer CHI_Buffer_receive[*cSlotIDMax*+1];

4.3 An Example of Communication System with FlexRay Model

4.3.1 Application Model

Automotive electronic control systems are comprised of hardware and software parts. The hardware includes electronic control unit, relevant interfaces, sensors, executive bodies and display mechanisms. Nodes can use different hardware units according to the requirements. The software is stored in ECU. The control function of ECU changes depending on the software importing, and the function of output model depend on the intended tasks. The software dominates the electronic control system to perform observation and control functions. Those electronic devices are connected by the network and nodes complete certain functions in a cooperative way.

FlexRay communication system is an automotive system based on the FlexRay communication protocols. When developing systems, the functional correctness of the ECU unit should be guaranteed, and the communication between nodes also has to meet the need of the FlaxRay protocols. More importantly, the whole system should be effectively implemented.

Name	Description	Description
cStaticSlotIDMax	Highest static slot ID number	2-1023
gCycleCounterMax	Maximum cycle counter value in a given cluster.	7-63
gdActionPointOffset	Number of macroticks the action point is offset form the beginning of a static slot	1-63MT
gdDynamicSlotIdlePhase	Duration of the idle phase within a dynamic slot	0-2 Minislot
gdMinislot	Duration of a minislot	2-63MT
gdMinislotActionPoint	Number of macroticks the minislot action point is offset from the beginning of a minislot	1-31MT
gdStaticSlot	Duration of a static slot	3-664 MT
gMacroPerCycle	Number of macroticks in a communication cycle	8-16000MT
gNumberOfMinislots	Number of minislots in thedynamic segment	0-7988
gNumberOfStaticSlots	Number of static slots in the static segment	2-1023
gPayloadLengthStatic	Payload length of a static frame	0-127 two-byte words
gdCycle	Length of the cycle	24us-16000us
gdMacrotick	Duration of the cluster wide nominal macrotick	1-6us
gdNIT	Duration of the Network Idle Time	2-15978MT
pPayloadLengthDynMax	Maximum payload length for dynamic frames	0-127 two-byte words
adActionPointDifference	Amount by which the static action point offset is greater than the minislot action point offset(zero if static slot action point is smaller thanminislot action point)	

Table 4.1: Configuration parameters of interface

Currently, we focus on the correctness of nodes communication, as well as the reachability and instantaneity. We extract the data of the nodes' input and output and establish the connection of nodes, according to the application design. Thus, we do not emphasise on the ECU operation and scheduling problems. In the ECU of every node, we consider tasks from the following aspects:

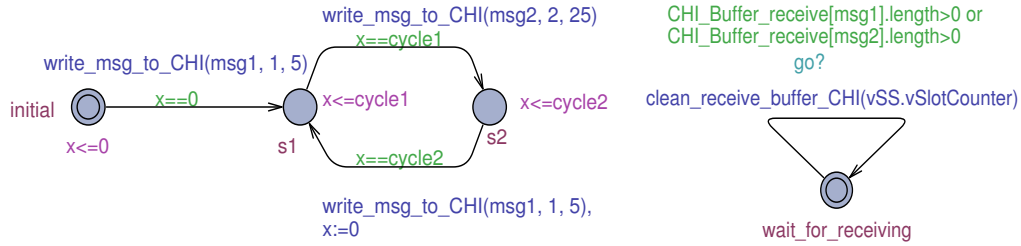
- We classify the task into 3 categories: the periodic, non-periodic and the mixed task.
- One ECU model can consider more tasks, which can be correlated with each other or not.
- To every single task, we can set worst case execution time (WCET) and best case execution time (BCET) in the model, or deadline.
- Two read and write functions are used to realize the input and output data of the task.

4.3.2 An Example of Communication System

Here, an example will be shown. This communication system has two ECUs, one is sender, the other one is receiver. The sender is two period tasks which send messages in specified slot. Each task has a executing time. And the receiver is responsible for receiving messages when receiving buffer has message. In the case of known the system function, the first step is setting communication cycle of FlexRay model. For this system, we assume parameters value as follow:

```
const int cSlotIDMax = 10;
const int cStaticSlotIDMax = 6;
const int gCycleCounterMax = 6;
const int gNumberOfStaticSlots = 6;
const int gNumberOfMinislots = 32;
const int gdCycle = 650;
const int gdStaticSlot = 5;
const int gdActionPointOffset = 2;
const int adActionPointDifference = 1;
const int gdMinislot = 3;
const int gdMinislotActionPointOffset = 1;
const int gdNIT = 4;
const int gdMacrotick = 5;
const int gdMacroPerCycle = 130;
const int pPayloadLengthDynMax = 200;
const int pPayloadLengthStatic = 200;
```

The next step is configuration of application design model. This system has only one message which allocated in the second slot to sending.



(a) (b)

Figure 4.2: (*a*) The sender model and (*b*) the receiver model in UPPAAL

```
const t_msg_slot msg1 = 2;
const t_msg_slot msg2 = 7;
```

The last step is using two function to read and write message in CHI. The functions are shown :

```
void write_msg_to_CHI(t_msg_slot msg, int value, int len)
  {
     CHI_Buffer_send[msg].data = value;
     CHI_Buffer_send[msg].length = len;
  };

void int read_msg_from_CHI(t_msg_slot msg)
  {
     return CHI_Buffer_receive[msg].data;
  };
```

Finally the application model is build as shown in Fig. 5.4. In the initial state of the sender, sender output the first message *msg1* to buffer of CHI after system start. Then it will wait for the sending cycle *cycle1*. The FlexRay model send this message in the second slot, and the message finally read by the receiver model from CHI. When the cycle time comes, the second message *msg2* will be stored in CHI, and it will be send in the slot7 of dynamic segment. After that, the sender waits for the second cycle *cycle2* and send *msg1* again. And then repeating this process. By this way, a communication system is set up under the FlexRay model in UPPAAL. After that, we can simulate and verify some properties of this system.

Chapter 5

Experiments

In this chapter, we will show three experiments. The first is a testing example which verifies the validity of the FlexRay model. The second is a simple application to verify the timing properties. The third one verifies the feasibility of the proposed framework using a practical automotive system and an adaptive cruise control system.

5.1 A Testing Example

The first test is a simple example, the basic structure of which is shown in Fig. 5.1. This communication cluster has four ECUs. Specifically, ECU1 and ECU2 send static messages, while ECU3 and ECU4 send dynamic messages. They are independent. One communication cycle has 10 slots in total, where 1 to 6 are static slots, and 7 to 10 are dynamic slots. We send messages with ECU1 using slots 1, 3, 5, ECU2 using slots 2, 4, 6, ECU3 using slots 7, 9 and ECU4 using slots 8, 10, respectively. In addition, all these nodes send data to a common receiver. Based on the testing case structure, the interface parameters of the test system are given as follows.

```
const int cSlotIDMax = 10;
const int cStaticSlotIDMax = 6;
const int gCycleCounterMax = 6;
const int gNumberOfStaticSlots = 6;
const int gNumberOfMinislots = 32;
const int gdCycle = 650;
const int gdStaticSlot = 5;
const int gdActionPointOffset = 2;
const int adActionPointDifference = 1;
const int gdMinislot = 3;
const int gdMinislotActionPointOffset = 1;
const int gdNIT = 4;
const int gdMacrotick = 5;
const int gdMacroPerCycle = 130;
```

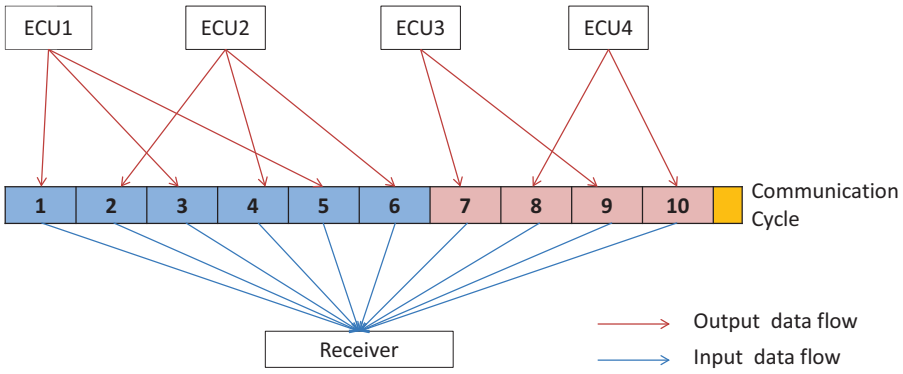


Figure 5.1: Testing example model in UPPAAL

```
const int pPayloadLengthDynMax = 200;
const int pPayloadLengthStatic = 200;
const t_msg_slot msg1 = 1;
const t_msg_slot msg2 = 2;
const t_msg_slot msg3 = 3;
const t_msg_slot msg4 = 4;
const t_msg_slot msg5 = 5;
const t_msg_slot msg6 = 6;
const t_msg_slot msg7 = 7;
const t_msg_slot msg8 = 8;
const t_msg_slot msg9 = 9;
const t_msg_slot msg10 = 10;
```

The ECU only has tasks for sending messages. These tasks check the sending buffer and if the allocated buffers are empty, the task will immediately send data to the supplement. The lengths of the output data of ECU1 and ECU2 are fixed. For ECU3 and ECU4, the output data lengths are also fixed, however, the values are selectable. This sort of design is because that the length of dynamic slots is changeable. The receiver only receive messages once the receiving buffer is not empty. ECUs and the receiver model in UPPAAL are shown in Fig. 5.1.

5.1.1 Verification

The communication system is verified through queries in UPPAAL.

- **Check1:** Is the system without a deadlock?
  Formula: A[] not deadlock

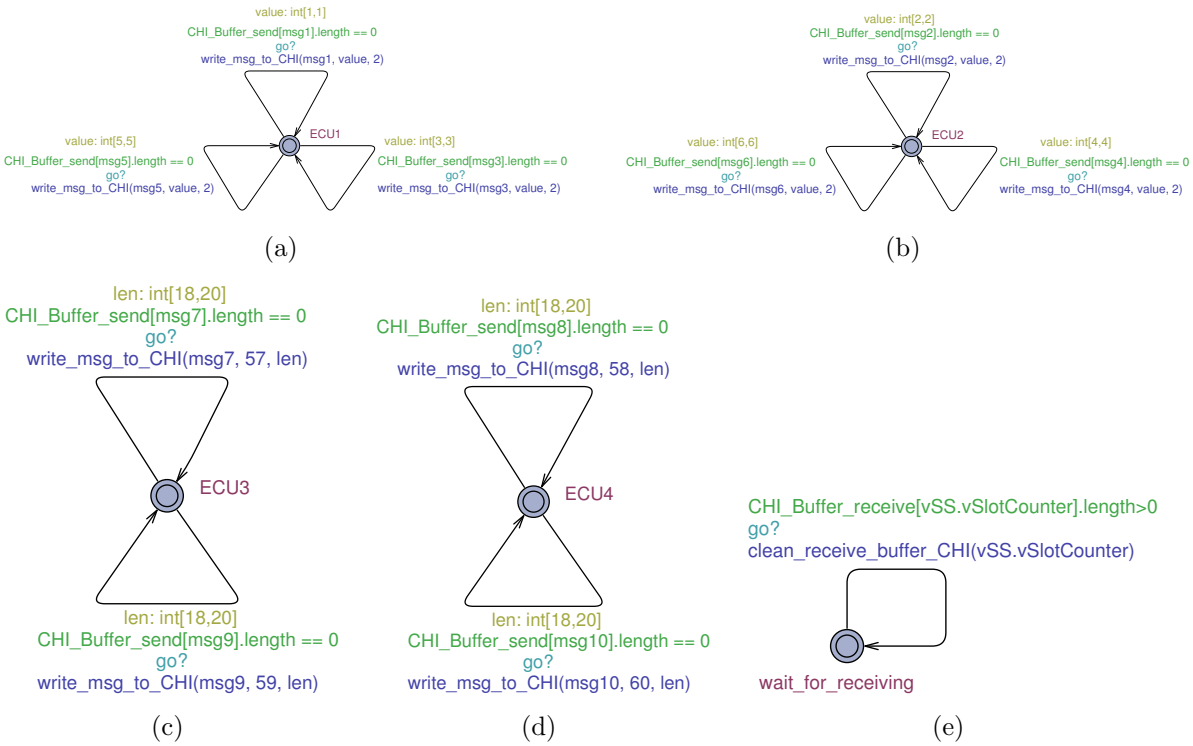


Figure 5.2: (*a*) ECU1 (*b*) ECU2 (*c*) ECU3 (*d*) ECU4 (*e*) Receiver

Fulfilled only if for all possible states in the system it is guaranteed that no deadlock status will occur. This shows FlexRay model can continuously send and receive messages.

- **Check2:** Is the message received in the allocated slot?
  Formula: *A*[] forall (i:int[1,10]) ((CHI_Buffer_receive[i].length>0) imply (vSS.vSlotCounter==i))

  Fulfilled if all messages are received in specified slot. If conditions are satisfied, when a receiver buffer in CHI has data coming in, the buffer ID and slot ID should be the same. It also show messages are sent in the allocated slot. The message sending/receving follows the slot ID ordering.

- **Check3:** Does only one buffer receive the transmitted message in the same slot?
  Formula: *A*[] forall (i:int[1,10]) forall (j:int[1,10]) (CHI_Buffer_receive[i].length>0 && CHI_Buffer_receive[j].length>0) imply (j==i)

  Fulfilled if only one buffer receives the message in the same slot. If two receiving buffers in CHI have data coming in at the same time, their IDs should be the same. It means only one node utilizes bus in every moment of the system.

- **Check4:** Does ECUs successfully output message?
  Formula: $E <>$ forall (i:int[1,10]) (CHI_Buffer_send[i].length>0)

  Fulfilled if FlexRay model can receive messages form ECUs, and ECUs transmitted message to CHI.

- **Check5:** Can the message of sending buffers be sent?
  Formula: forall (i:int[1,10]) ((CHI_Buffer_send[i].length>0) imply (CHI_Buffer_send[i].length==0))

  If this query is satisfied, the FlexRay model can send message from CHI.

- **Check6:** Is it possible all messages are received by the receiver?
  Formula: $A <>$ CHI_Buffer_receive[msg1].length > 0

  Only fulfilled if the msg1 is received eventually by receiver. Similarly, this query can also verify other messages, such as $A <>$ CHI_Buffer_receive[msg2].length > 0 and so on. Because the limitation of UPPAAL, messages need to be verified respectively.

5.1.2 Evaluation

Firstly, Check1 confirms that this communication system is not deadlock, and the system can normally operate. Then, Check2 and Check3 are satisfied in this experiment, which means that the communication capabilities are in accord with TDMA and FT-DMA schemes of the FlexRay protocol. In addition, the FlexRay model achieves the basic communication requirements. Messages are normally transmitted in static and dynamic segment. Moreover, the FlexRay model sends and stores messages in correct slots, where the configuration of interface is effective for ECUs. Thirdly, Check4, Check5 and Check6 verify whether the proposed interface is viable to connect ECUs and FlexRay model. Finally, ECU can be successfully realized sending and receiving messages with FlexRay model.

The main purpose of this experiment is to verify the validity and usability of the FlexRay model and its interface. For the testing example, Check1 to Check6 are fulfilled. FlexRay model can handle sending and receiving messages between nodes in the communication system according to FlexRay communication protocols. However, the timing properties of the FlexRay system is very important with verification requirements. In the next section, the example of an practical system will be discussed.

5.2 Response Time Checking with FlexRay Model

In this experiment, the response time of a simple application model is checked. There is a sender and a receiver in this model, as shown in Fig. 5.3. The sender works cyclically, and

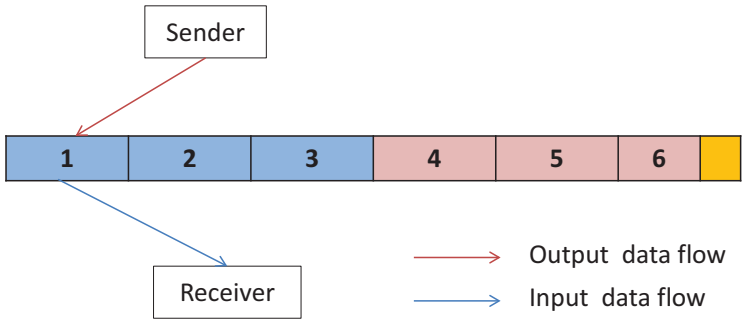


Figure 5.3: The structure of a simple application model

it sends a testing message right after the system starts. When a cyclic time is reached, the sender will check whether this message has been sent out. If successful transmission is confirmed, then the next message will be sent. Otherwise, the sender will have to wait for the next cycle until the current message is confirmed to be successfully transmitted. The receiver receives data from the CHI buffer and also gives the responses. The parameters of the communication cycle of this system are set as follows:

```
const int cSlotIDMax = 6;
const int cStaticSlotIDMax = 3;
const int gCycleCounterMax = 6;
const int gNumberOfStaticSlots = 3;
const int gNumberOfMinislots = 30;
const int gdCycle = 910;
const int gdStaticSlot = 10;
const int gdActionPointOffset = 1;
const int adActionPointDifference = 0;
const int gdMinislot = 5;
const int gdMinislotActionPointOffset = 2;
const int gdNIT = 2;
const int gdMacrotick = 5;
const int gdMacroPerCycle = 182;
const int pPayloadLengthDynMax = 200;
const int pPayloadLengthStatic = 200;
const t_msg_slot msg1 = 1;
```

The data transmitted from the sender will be allocated to the first slot of the static segment. The operation cycle of sender is set at 100 macroticks (MT). The sender and receiver model are shown in Fig. 5.4.

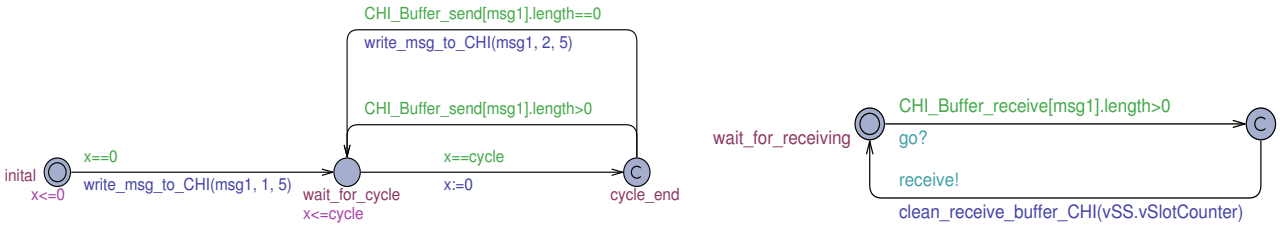


Figure 5.4: (a) The sender model and (b) the receiver model in UPPAAL

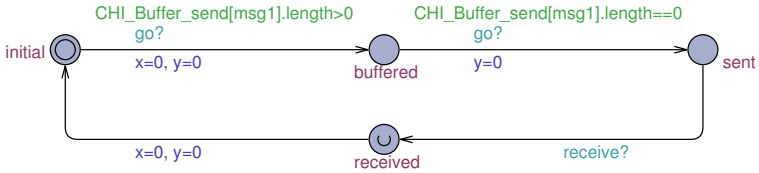


Figure 5.5: Observer model

5.2.1 Verification

To verify the system response time, we build a observer model as shown in Fig. 5.5, to check the transmission state. According to the state change, the best case response time (BCRT) and the worst case response time (WCRT) can be calculated. When the message is sent to the send buffer from the sender, the observer starts timing. The data are confirmed to be sent out when the send buffer is empty. The receiver will inform the observer after receiving the data, and then the observer will mark that this message has been successfully transmitted. One system response has finished so far, and the response time equals to the value of x in the received state. In the buffer state, y is the waiting time of this message. While in the sent state, y is the time required for the transmission of this message.

The WCRT and BCRT are checked by using a group of queries as shown in Tab. 5.2.1.

Property	Result
Check1: E<> (observer.received && observer.x > 182)	No
Check2: E<> (observer.received && observer.x == 182)	Yes
Check3: A[] (observer.received imply observer.x <= 182)	Yes
Check4: A[] (observer.received imply observer.x <= 181)	No
Check5: E<> (observer.received && observer.x < 6)	No
Check6: E<> (observer.received && observer.x == 6)	Yes
Check7: A[] (observer.received imply observer.x >= 6)	Yes
Check8: A[] (observer.received imply observer.x >= 7)	No

Table 5.1: Results of the experiments

Check1 to Check4 indicate that the time from buffered to the received of observer is 182 MT. Check5 to Check8 verify that the BCRT is 6 MT. In addition, Check9 (A[] (observer.received imply observer.y == 5)) can be satisfied. It means sending of *msg1* always costs 5 MT, which is the message length.

5.2.2 Evaluation

From this example, the timing property of the system is tested by using observer model. In the case when the parameters are fixed, a group of BCRT and WCRT can be obtained. According to our experience, if the configuration changes, for instance the message is allocated to other slot to transmit, different BCRT will be got. The message sent in the slot with smaller ID will has the fastest response. On the other hand, the settings of the application also affect the testing result, such as the operation cycle. Hence, both the application and FlexRay should be considered for the timing property of the system.

5.3 Adaptive Cruise Control Subsystem

Due to the fact that the cruise control subsystems can make driving more comfortable and safer, much attention has been drawn among researchers and auto makers. One of them which has been widely studied and accepted is the adaptive cruise control system (ACC) and it can keep the distance between the self car and the one in front, by controlling the car's accelerated velocity automatically, in order to improve the active safety of the vehicles. Adaptive cruise control is an optional cruise control system appearing on some more luxury vehicles. The BMW E90 new 3 series have been equipped with a driving assistant system (ACC), provided by Bosch, Germany.

ACC derives from the conventional cruise control system, and it mainly comprised of ranging sensor, ECU and actuator. The ranging sensor is basically a radar, which measures the relative distance, velocity and acceleration of the self car and the one in front. The ECU calculates and controls the speed and acceleration to set for keeping the front distance of the car. ECU also sends control commands to the relevant executive bodies. The actuator consists of the actuator and the brake actuator, which aim to adjust the acceleration of the vehicles as required. The ECU diagram with ACC being using in FlexRay bus is shown in Fig. 5.6.

ECU1 periodically receives information data from two radar sensors. The information data coming from each radar is processed by an object detection task operating on ECU1. The processed data are transmitted in the bus to a data fusion task operating on ECU2, as well as object selection and adaptive cruise control tasks. Then the data stream is sent via the bus again to ECU3 running on the other two tasks and the final output is sent to an actuator, over the same bus. All the information data are mapped onto the dynamic segment of the FlexRay bus.

Tab. 5.3 provides the relevant parameters value of the system [7]. The length of messages equals to the number of minislots. This is WCET of each task. Then we need to transform millisecond to macrotick or minislot. According to this paper, we know that

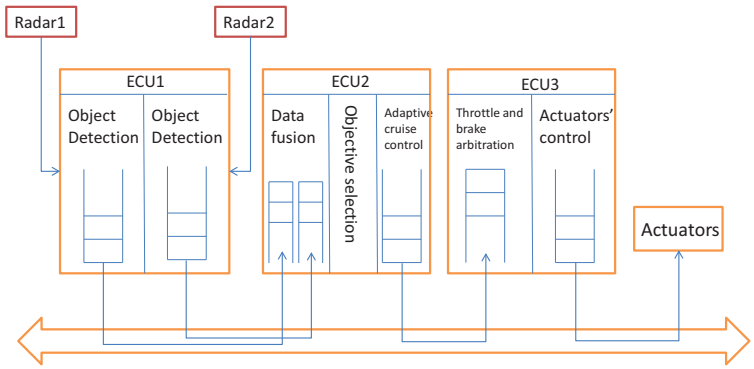


Figure 5.6: Adaptive cruise control system

Bus		ECUs	
Message	♯Minislot	Tasks	WCET
msg1	64	Data Fusion	19.7 ms
msg2	64	Object Selection	1 ms
msg3	15	Adaptive Cruise Control	4.3 ms
msg4	40	Arbitration	17.6 ms
		Actuator control	9 ms
		Object detection	12.5 ms

Table 5.2: The relevant parameters value of the adaptive cruise system

a communication cycle is 10 ms and DYN segment is 8 ms, consisting of 72 minislots. Therefore, we can compute a minislot and macrotick duration, which equals to the number of milliseconds. In UPPAAL, the time unit is macrotick. However, these parameter values are very large and moreover they are not integers. In order to shorten the time and simplify the conditions of checking, we proportionally reduce the numerical values.

Due to that all messages in this system are sent in dynamic segment, we allocate slots for ECU, which is shown in Fig. 5.7. ECU1 sends m1 and m2 to ECU2 in slot3 and slot4, respectively. ECU2 send m3 to ECU3 in slot5 and ECU3 send m4 to actuator in slot6.

According to the structure of ACC and the relevant parameters, the interface between FlexRay model and ECU are set as follows:

```
const int cSlotIDMax = 5;
const int cStaticSlotIDMax=2;
const int gCycleCounterMax=5;
const int gNumberOfStaticSlots = 2;
const int gNumberOfMinislots = 36;
const int gdCycle= 2500;
const int gdStaticSlot=20;
const int gdActionPointOffset=2;
```

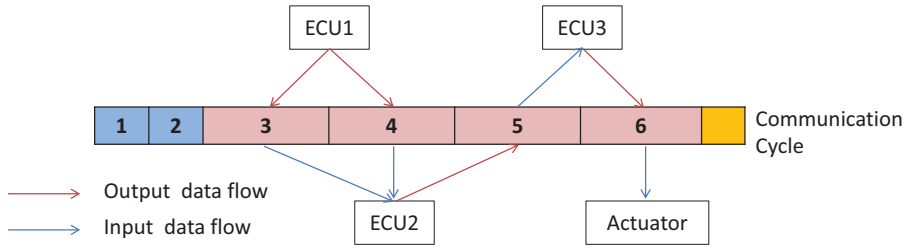


Figure 5.7: The communication cycle of adaptive control system

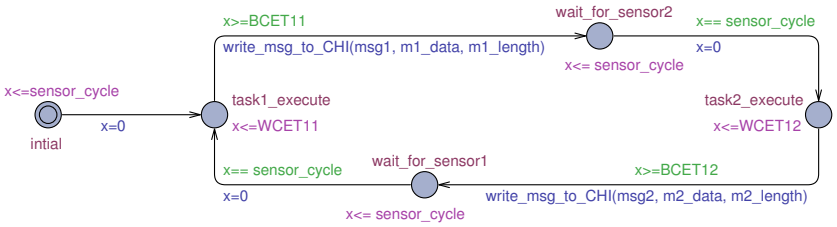


Figure 5.8: The ECU1 model of ACC

```
const int gdMinislot=10;
const int gdMinislotActionPointOffset=2;
const int adActionPointDifference=0;
const int gdNIT = 2;
const int gdMacrotick =5;
const int gdMacroPerCycle=410;
const int pPayloadLengthDynMax =200;
const int pPayloadLengthStatic =200;
const t_msg_slot msg1 = 2;
const t_msg_slot msg2 = 3;
const t_msg_slot msg3 = 4;
const t_msg_slot msg4 = 5;
```

5.3.1 Verification

For the verification of the ACC system, the following verification queries were applied.

- **Check1:** Is the system without a deadlock?
  Formula: A[] not deadlock.

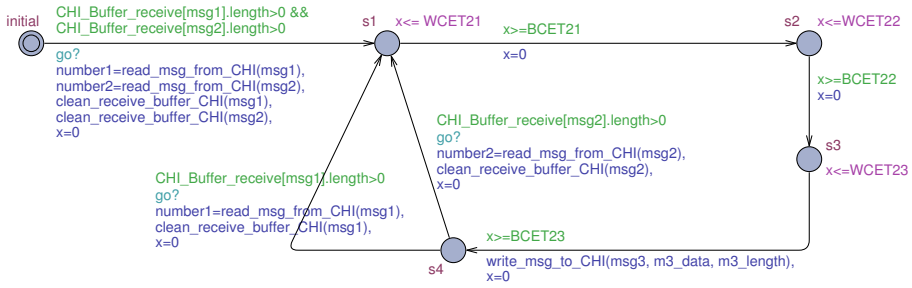


Figure 5.9: The ECU2 model of ACC

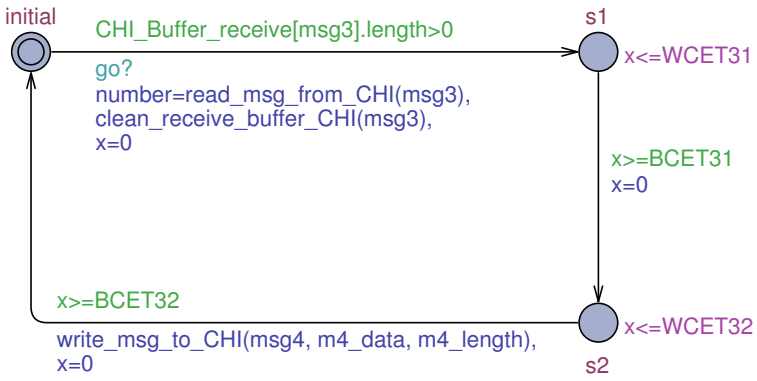


Figure 5.10: The ECU3 model of ACC

Fulfilled only if for all possible states in the system it is guaranteed that no deadlock status will occur. This shows whether the system is valid .

- **Check2:** Do sensors send the message can get a response?
  Formula: $A <>$ CHI_Buffer_receive[msg4].length>0

This query checks the basic requirements of the system. Eventually, the actuator can receive a message for adjustment of motor, which check the reachability of messages in the system.

5.3.2 Evaluation

This experiment mainly verifies the feasibility and reachability. Check1 is satisfied. It shows that FlexRay model can be used to determine the practical application of automotive control systems, although this model has certain limitations. When the node is excessive or the variable is very complex, checking query will be a long time as well as memory overflow.

Check2 verifies that the ACC system can respond eventually. The actuator can receive messages and responses. In order to guarantee the reachability of messages in the design model, we need to adjust the configuration of the communication cycle and the design of ECUs. In the process of adjusting, we know that this property is not satisfied, when the number of minislots is less than or equal to 20. This fully shows the flexray flexibility and real-time. In the dynamic segment, the number of minislots is essential in the configuration. If the message lengths are all very large, the messages with smaller ID number will not be sent. This problem can be solved by increasing the length of the dynamic segment. The experience is useful for determining the minimum number of minislots for an application. On the contrary, if the data lengths in static segment are mostly small, we should reduce the static slot length to save the bandwidth.

Although we can easily obtain the message response time in the previous example, verification would consume much longer time for ACC system, as well as more RAM for a complicated communication system. Additionally, there are many ECUs within one system having different characteristics, respectively, therefore it is difficult to analyze their temporal properties. Moreover, the state-space explosion problem has also be considered. Hence, there is a limit of system scale in the FlexRay model, and one easy solution is to divide the whole system into several parts and perform model checking separately. However, this approach can not reflect the communication flexibility of ECUs controlled by the FlexRay bus protocols. Hence, in the future work, the checking framework has to be modified and improved based ont the FlexRay model.

Chapter 6

Conclusion

6.1 Summary

In this work, an UPPAAL framework to test and verify FlexRay communication systems has been proposed. The main features of the proposed approach are summarized as follows:

The framework provides an easy-to-use method in UPPAAL for model checking designs of the communication system using FlexRay bus. The FlexRay model is the core of the framework, and it does not need to change to verify different systems. We only need to set up of the parameters of the interface, and the communication system can be connected to the FlexRay model.

We showed two examples to explain how the framework can be used. Through the experiments, the reachability and the feasibility can be checked. Also, we checked time related properties, such as the response time of message and the response time of communication system. During the experiment, we showed how to verify the system, how to modify the configuration of interface to optimize the system and ECUs, and ultimately to achieve the best response time.

6.2 Future Work

At present, our FlexRay model is not perfect. With regard to the FlexRay communication protocol itself, it also has many features that have not yet been implemented, such as the synchronization and start up processes. FlexRay communication system has a lot of the time parameters. When we check a certain timing properties of application, the process of adjusting time parameters has many changes. We still need experiments to study the influence of the parameters of the system. In addition, the verification of the large-scaled and complex system will cost time and the response time is hard to get because the ECUs and transmitting message is complex. Therefore, it is better to further simplify the model in order to save the verification time.

Bibliography

[1] 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.

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

[3] Y.-N. Xu, Y. Kim, K. Cho, J. Chung, and M. Lim, "Implementation of Flexray Communication Controller Protocol with Application to a Robot System," in *Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on*, pp. 994 –997, 31 2008-sept. 3 2008.

[4] M. Short and M. Pont, "Hardware in the loop simulation of embedded automotive control system," in *Intelligent Transportation Systems, 2005. Proceedings. 2005 IEEE*, pp. 426 – 431, sept. 2005.

[5] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei, "Timing analysis of the Flexray communication protocol," in *Real-Time Systems, 2006. 18th Euromicro Conference on*, pp. 11 pp. –216, 0-0 2006.

[6] K.-H. Jung, M.-G. Song, D. ik Lee, and S.-H. Jin, "Priority-based scheduling of dynamic segment in Flexray network," in *Control, Automation and Systems, 2008. ICCAS 2008. International Conference on*, pp. 1036 –1041, oct. 2008.

[7] A. Hagiescu, U. Bordoloi, S. Chakraborty, P. Sampath, P. Ganesan, and S. Ramesh, "Performance Analysis of FlexRay-based ECU Networks," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 284 –289, june 2007.

[8] C.-C. Wang, G.-N. Sung, C.-L. Wang, P.-C. Chen, M.-F. Luo, and H.-C. Hu, "Physical layer design for ECU nodes in FlexRay-based automotive communication systems," in *Consumer Electronics, 2009. ICCE '09. Digest of Technical Papers International Conference on*, pp. 1 –2, jan. 2009.

[9] C. Muller, M. Valle, R. Buzas, and A. Skoupy, "Mixed-mode behavioral model of FlexRay physical layer transceiver," in *Circuit Theory and Design, 2009. ECCTD 2009. European Conference on*, pp. 527 –530, aug. 2009.

[10] 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.

[11] K. Klobedanz, A. Koenig, W. Mueller, and A. Rettberg, "Self-Reconfiguration for Fault-Tolerant Flexray Networks," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pp. 207 –216, march 2011.

[12] T. Hiraoka, S. Eto, O. Nishihara, and H. Kumamoto, "Fault tolerant design for X-by-wire vehicle," in *SICE 2004 Annual Conference*, vol. 3, pp. 1940 –1945 vol. 3, aug. 2004.

[13] J. N. Jan Malinsky, "Verfication Of Flexray Start-up Mechanism by Timed Automata," in *Metrology and Measurement Systems*, vol. XVII, pp. 461 –480, Sep. 2010.

[14] A.Albert, "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," *In Embedded Word, Nurnberg, Germany*, http://www.semiconductors.bosch.de/pdf/embedded_word_04_albert.pdf 2004.

[15] G. sheng Feng, W. Zhang, S. me Jia, and H. sheng Wu, "CAN Bus Application in Automotive Network Control," in *Measuring Technology and Mechatronics Automation (ICMTMA), 2010 International Conference on*, vol. 1, pp. 779 –782, march 2010.

[16] Y. Xu, J. Wang, W. Chen, J. Tao, and Q. Liu, "Application of LIN Bus in Vehicle Network," in *Vehicular Electronics and Safety, 2006. ICVES 2006. IEEE International Conference on*, pp. 119 –123, dec. 2006.

[17] W. S. Kim, H. A. Kim, J.-H. Ahn, and B. Moon, "System-Level Development and Verification of the Flexray Communication Controller Model Based on SystemC," in *Future Generation Communication and Networking, 2008. FGCN '08. Second International Conference on*, vol. 2, pp. 124 –127, dec. 2008.

[18] "The Flexray Communication System Specification Version 3.0.1," http://www.flexray.com.

[19] B. F. H.-J. P. Michael Gerke, Rdiger Ehlers, "Model checking the flexray physical layer protocol," in *Formal Methods for Industrial Critical Systems*, pp. 132–147, 2010.

[20] 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.

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

[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] Z. Xin-feng, W. Jian-dong, L. Bin, Z. Jun-wu, and W. Jun, "Methods to Tackle State Explosion Problem in Model Checking," in *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, vol. 2, pp. 329 –331, nov. 2009.

[24] D. L. Rajeev Alur, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, April 1994.

[25] R. Alur and D. Dill., "Theoretical Computer Science," *Automata for Modelling Real Time Systems*, pp. 132–147, 1994.

[26] S. Yovine, "KRONOS: averification tool for real-time systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, pp. 123–133, 1997.

[27] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pp. 125 –126, sept. 2006.

[28] 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.

[29] A. D. Gerd Behrmann and K. G.Larsen, "A Tutorial on UPPAAL 4.0," vol. http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, November 2006.

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