The Journal of Systems and Software 161 (2020) 110461

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Model checking of in-vehicle networking systems with CAN and FlexRay☆

Xiaoyun Guo[a,*], Toshiaki Aoki[a], Hsin-Hung Lin[b]

[a] School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Ishikawa, Japan
[b] Institute of Information Science, Academia Sinica, Taipei, Taiwan

ARTICLE INFO

Article history:
Received 22 February 2019
Revised 9 October 2019
Accepted 11 November 2019
Available online 12 November 2019

Keywords:
Model checking
In-vehicle networking systems
UPPAAL
CAN
FlexRay

ABSTRACT

An in-vehicle networking (IVN) system consists of electronic components that are connected by buses and communicate through multiple protocols according to their requirements. In practice, intelligent vehicles need to exchange data between subsystems that use various protocols, such as the controller area network (CAN) and FlexRay. Such systems are more likely to encounter delays and message loss during transmission, presenting serious safety issues. Moreover, IVN systems are extremely complicated because of their large number of nodes, multiple communication protocols, and diverse topologies. As a result, it is difficult to check the timed properties of the system directly and accurately. In this paper, we present an appropriate abstraction for modeling IVN systems that utilize CAN and FlexRay during the design phase. The timed properties of communication are analyzed using the UPPAAL platform. As there are numerous IVN system structures, a framework is developed to build a model for an IVN system design. The model is to verify the transmission of messages between different protocols. We evaluate the validity, applicability, and reusability of the framework and show the performance of the framework for verifying IVN system models.

1. Introduction

The modern automobile industry has replaced most of the traditional mechanical and hydraulic systems with the X-by-Wire technique (Hiraoka et al., 2004). The X-by-Wire technique is abstracted from the Fly-by-Wire technique in airplanes, where "X" refers to a specific electronic control system, such as in Break-by-Wire or Steer-by-Wire (Armbruster et al., 2006). Such systems utilize electronic control units (ECUs) as controllers rather than using mechanical components. These controllers process data detected from sensors and transmit results (electronic control signals) to actuators, which perform control operations. Between the sensors, controllers, and actuators, the data are transmitted by buses. The X-by-Wire technique greatly simplifies the structure of a given vehicle and improves the response time and flexibility of controls (Waszniowski et al., 2009). To meet security, comfort and entertainment requirements, electronic control systems become increasingly diverse in vehicles, and they constitute an in-vehicle networking (IVN) system (Coles et al., 2016). An IVN system is a real-time distributed system consisting of multiple subsystems, each of which employs different communication protocols to realize data transmission according to its requirements (Grimm, 2003; Demmeler and Giusto, 2001). The controller area network (CAN), local interconnect network (LIN), FlexRay, media-oriented systems transport (MOST) and other protocols are applied in IVN systems (Leen and Heffernan, 2002; Wolf et al., 2004). CAN is a field-bus with a wide range of application areas, which has good fault tolerance and is often used in the powertrain system and body control system of a vehicle (Johansson and Martin, 2005). LIN is a simple low-cost bus used as a complement to other buses and is suitable for controlling seats, air-conditioning, lights, and so on (Elmenreich and Krywult, 2005). FlexRay is a high-speed, flexible and reliable communication protocol aimed at IVN systems, and is applicable to control systems with high requirements for real-time responsiveness, security, and reliability (Park and Sunwoo, 2011; Dominguez-Garcia et al., 2006). MOST was developed for in-car multimedia systems with high bandwidth, and is used in navigation, televisions, CD players and other entertainment systems (Wolf et al., 2004). These subsystems, which use different pro-

☆ This paper is an extension of Guo et al. (2017).

* Corresponding author.
E-mail addresses: xiaoyun@jaist.ac.jp (X. Guo), toshiaki@jaist.ac.jp (T. Aoki), hlin@iis.sinica.edu.tw (H.-H. Lin).

https://doi.org/10.1016/j.jss.2019.110461
0164-1212/

tocols, connect to a central gateway to transfer data (Seung-Han et al., 2008).

Nowadays intelligent vehicles need to exchange safety data frequently between the body control subsystem and the chassis control subsystem (Baronti et al., 2011). Such IVN systems are directly related to safety, it is vitally important to verify communication between different protocols in the IVN system. There are some works that checked IVN systems based on the integration platform (Sangiovanni-Vincentelli, 2003; Demmeler and Giusto, 2001; Feng et al., 2010). They implemented the system design on the platform and did system testing. Testing is using test cases as input to check the correctness of the output of the system. But it is difficult to create a set of cases to detect all states of the system. Besides, there are several existing works that checked the IVN system with a single protocol using a model checking technique, such as (Waszniowski et al., 2009; Pan et al., 2014). However, an IVN system employs multiple protocols for transmitting messages, and these protocols have specific communication mechanisms, which may influence each other, as we illustrate the example in Section 2.3. The results of checking the whole IVN system may differ from subsystems when separately checked.

Thus, we were motivated to develop a framework for modeling and verifying IVN systems during the design phases using timed model checking techniques. Model checking technique can exhaustively check properties by searching all states of the system. The framework is applied to verify IVN system designs with both CAN and FlexRay. We focus on the timed property of communication between different protocols, as this is critical to the safety of an IVN system. To analyze the timed property, we describe the modeling and verification of IVN system design models using the UPPAAL platform.

This work attempted to challenge two issues: 1) to find an appropriate abstraction for modeling complex IVN systems, 2) to propose a reusable framework to apply to a variety of IVN systems.

Firstly, a practical IVN system may consist of hundreds of nodes, several communication protocols, and gateways for bridging between the different protocols. These nodes and protocols also contain details related to applications, protocol mechanisms, and hardware implementations. Modeling every detail of such a complex system would make it impossible to verify the functionality because of the state space explosion problem (Zhu et al., 2009). Therefore, it is necessary to abstract the IVN system to make the state space as small as possible. However, a model that is too abstract may not be able to verify the timed properties of the system accurately. Because an appropriate abstraction is essential to achieve the goal of this study, there must be a trade-off between accuracy and abstraction. Although several studies have discussed the modeling of the CAN and FlexRay protocols separately (Waszniowski and Hanzálek, 2008; Saha and Roy, 2007), there has been little research on IVN systems operating with both. A system in which different types of nodes co-exist (i.e., CAN nodes, FlexRay nodes, and gateways) will operate under multiple protocols, which make the abstraction non-trivial. Thus, to overcome this issue, we adopt a two-staged strategy to derive abstractions for both the system architecture and communication protocols.

Secondly, real IVN systems have various topologies of connections between nodes. A model for an IVN system with a particular topology is unable to check other IVN systems with different topologies, limiting the applicability of the framework. Based on the two-stage abstraction strategy, we propose a framework that adapts to different topologies. This framework is composed of a protocol module, interface module, medium module, configuration module, forwarder module, and environment module. The interface, medium, configuration, and protocol modules are reusable, enabling IVN system design models with different topologies to be constructed by modifying the appropriate parameters in the configuration module. Using the UPPAAL platform, this framework is validated by checking the communication behavior against the protocol specifications. Based on the proposed framework, we construct design models with three typical topologies: central, backbone, and daisy chain. By checking a series of timed properties, the framework provides estimates of the response time of messages in the system design. Finally, the reusability of this framework over different IVN system designs was evaluated by comparing the source code in UPPAAL, and we show the performance of the framework for verifying IVN systems.

This paper is an extension of our work reported in Guo et al. (2017). In particular, we abstract a common structure of CAN and FlexRay to propose a reusable framework in Section 3. A UML class diagram is presented to formally describe the framework in Section 4. We add more details to present each module of the framework in Section 4. In addition, the applicability, reusability, and performance of the framework will be illustrated by experiments in Section 5.

The remainder of this paper is organized as follows. In Section 2, we discuss the background to this work. The abstraction strategy and the framework are then introduced in Sections 3 and 4. Section 5 demonstrates the validity of the framework by checking properties based on protocol specifications and shows how to check the reachability and response time of messages. Also, we evaluate the applicability and reusability of the framework and discusses the performance of the framework. Section 6 gives a concise review of related work, before Section 7 concludes this paper and outlines ideas for future work.

2. Background

2.1. Controller area network

The CAN protocol provides efficient and secure support for data communication in distributed real-time control systems (Bosch, 1991). Its application domain ranges from high-speed networks to low-cost multiplex wiring. In the automotive industry, CAN is used for mainstream powertrain communication systems with bitrates up to 1 Mbit/s and low-cost body control systems (Seo and KIM, 2013). The CAN protocol adopts multi-master broadcasting and transfers messages based on the event-trigger algorithm. The CAN data frame consists of fields of 0–8 bytes. To ensure that only one node accesses the bus at a time, the CAN protocol uses a collision avoidance mechanism to arbitrate between messages. Each message has an identifier denoting its priority, and the message with the highest priority is sent once the bus is free. Other messages wait for the next arbitration period.

Fig. 1 shows an example of the message transmission scheme in CAN. There are two messages, `m1` and `m2`, in the transceiver buffer, and the bus is idle at time `x`. The arbitration process determines that `m1` has the highest priority, and so this message is immediately sent to the bus. Once `m1` has been sent, no other message

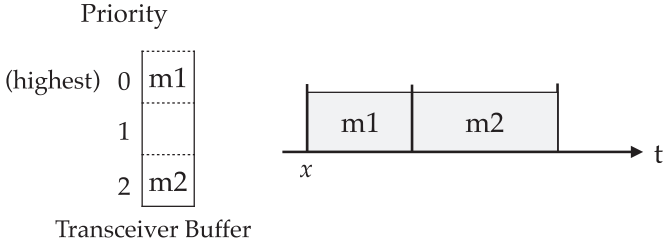


Fig. 1. Example of CAN message transmission.

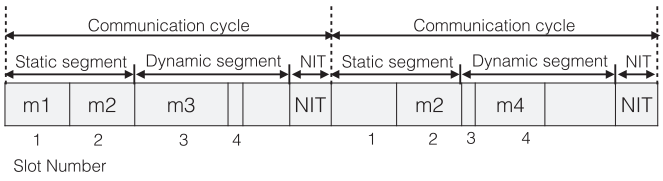


Fig. 2. Example of FlexRay message transmission.

is stored in the buffer. In the next arbitration period, m2 has the highest priority and is sent to the bus.

2.2. FlexRay

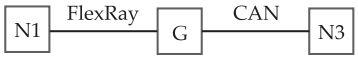
FlexRay is a high-speed, flexible communication protocol for data rates of up to 10 Mbit/s, and offers excellent fault-tolerance computing (Altran Tecnologies). This protocol is widely used for data communication in safety-critical fields in the vehicle, such as the chassis control networking system and the adaptive cruise control system (Kim et al., 2015). The FlexRay protocol adopts multi-master broadcasting and supports two media access schemes: a static time division multiple access (TDMA) scheme and a dynamic mini-slotting-based scheme (Altran Tecnologies). FlexRay data frames contain fields of 0–254 bytes. Message transmission takes place in periodic communication cycles, each of which includes a static segment and a dynamic segment. The static segment has a configurable number of static slots and is based on the unique assignment of message identifiers to transmit messages in each of these slots. The dynamic segment has a configurable number of mini-slots and is based on a fixed priority scheme that assigns priorities to the message identifiers. The length of a dynamic slot depends on the length of the message, that is, the number of mini-slots. An action point in each static slot and the first mini-slot of a dynamic segment determines the start of the transmission. The remaining mini-slots are assigned to network idle time (NIT) in that cycle.

Fig. 2 demonstrates two communication cycles for representing FlexRay message transmission. Each cycle has four slots, and the message identifier is set to the slot number for convenience. Messages, m1 and m2, are assigned to static slots, slot1 and slot2. Messages m3 and m4 are assigned to dynamic slots, slot3 and slot4, while m1 and m2 should be sent in slot1 and slot2. Although m1 is not sent in the second cycle, the time interval of slot2 still elapses with no transmission. m3 and m4 should be sent in slot3 and slot4, and the length of these dynamic slots depends on the length of the message. m4 is not sent in the first cycle, but slot4 occupies a time interval of one mini-slot. When the maximum number of slots is reached but the maximum number of mini-slots is not, there are no more transmissions until the end of that dynamic segment. FlexRay is thought to be the future of safety control systems in the automotive industry because of its high speed and flexibility.

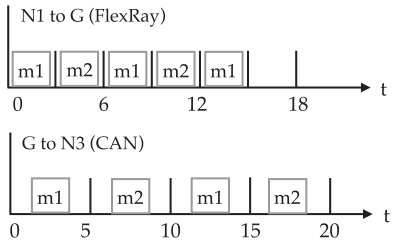
2.3. IVN systems in the presence of CAN and FlexRay

An intelligent vehicle adopts multiple protocols and frequently wishes to exchange safety data between them. For example, the CAN body control subsystem must communicate safety data to the FlexRay chassis control subsystem (Baronti et al., 2011). Hence, it is extremely important to verify the effectiveness of communication between CAN and FlexRay. For such systems, although each subsystem may be checked separately, communications between subsystems and the impacts of different protocols on the system are often ignored. In practice, different protocols have different communication modes, and their interaction may not satisfy the timed properties of the IVN system. In real-time systems, the message transmission time is important and will be subject to time constraints. The timed property typically reflects the transmission time from a source node to a destination node, referred to as the response time in this paper.

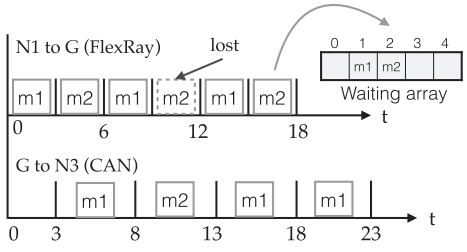
There is a simple communication system with CAN and FlexRay for clarifying the above problem. This system consists of three nodes, N1 follows FlexRay protocol to transmit messages, N2 follows CAN protocol to transmit messages and G is a gateway connecting N1 and N2. The topology of the system is shown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



(a) The structure of the system.



(b) Transmission time for FlexRay and CAN.



(c) Transmission time for the whole system.

Fig. 3. Message response times.

they alternately reach the G node. The transmission times of these messages are identical, both being 3-time-unit. In the CAN part, referring to the transmission result of the FlexRay part, m1 and m2 are transmitted in turn and the transmission times of these messages are 5-time-unit. As for the message transmission time in the whole system, we can combine the two results and it is 8-time-unit.

When we observe the whole system in a timeline, the transmission time of the messages is shown in Fig. 3 (c). As the messages are sent from N1 to N2, the transmission time from N1 to G under FlexRay will affect the transmission time from G to N2 by CAN. From N1 to G, the transmission time is the same as Fig. 3 (b). After the message reached the G node, there is a waiting array to store received messages in the G node, and the G node will forward messages to the CAN network. When the first m1 is received and stored in the array in accordance with its priority, the time is 3. Instantly, G sends the message to the CAN bus, and it will be received at t=8. While t=8, the first m2 was already waiting in the array. Hence, the m2 is transmitted immediately, and at $t = 13$, the message will be obtained by N2. At that moment, the FlexRay part of the system has finished the second communication cycle, and the second m1 and m2 existed in the array. Since the priority of m1 is the highest one, the G node sends the second m1 to N2 first. But when the transmission of the message ends in CAN, that is t=18, the third communication cycle also ends. The second m2 that was waiting to be sent has been overwritten by the new message. In the whole system, the G node periodically receives the m1 and m2 according to the 6-time-unit communication cycle of FlexRay. All m1 can be successfully sent to N2 because of the higher priority, but some m2 that wait for more than 6-time-unit in the array will be lost. Additionally, the transmission time of every m1 is different. For example, the first m1 took 8-time-unit and the second m1 took 12-time-unit from N1 to N2. If there is a timed property that demands the transmission time for all messages is less than 10-time-unit, the property is unsatisfied in considering multiple protocols. As a result, the checking result of the timed property of the IVN system will be affected by different communication mechanisms. Hence, the timed properties of the communication between multiple protocols should be considered.

2.4. UPPAAL

UPPAAL is an integrated tool environment for the modeling, simulation, verification of real-time systems modeled as networks of timed automata. It serves as a modeling/design language that precisely describes behavior under time constraints. UPPAAL can check the invariant, reachability, and liveness properties specified in a fragment of computational tree logic (CTL) (Bengtsson et al., 1995). Specifically, with an observer model that monitors frame transmissions, the response time can be obtained and the timed properties can be checked. UPPAAL exhaustively searches through the system state space to investigate the desired properties. Some case studies have used UPPAAL to examine communication protocols, real-time systems, and multimedia applications (Bel Mokadem et al., 2005; Pan et al., 2014).

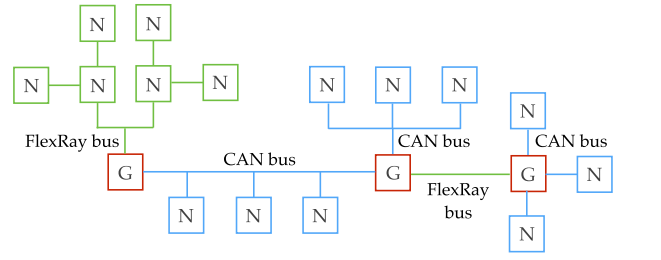
3. Abstraction

Model checking is an effective method on verification of safety-critical systems. It exhaustively and automatically verifies properties by searching all states of the system. The properties can be specified in temporal logic and verified precisely. However, the capability of model checking tools is limited due to the state explosion problem. An IVN system is not only composed of multiple nodes, but each node also needs to conform to a corresponding communication protocol specification and operating system standards. The number of system states is astronomical. Furthermore, these specifications and standards have a number of parts that are irrelevant in terms of verifying the communication behavior of IVN design models. Hence, it is necessary to make an appropriate abstraction for IVN systems. The abstraction is an indispensable method for reducing the state explosion problem (Zhu et al., 2009). We simplify the system for properties that need to be verified, keeping the parts related to it. The abstracted system can effectively check the properties while minimizing the number of system states. We give a two-staged abstraction, abstracting the architecture of the system and the composition of protocol specifications.

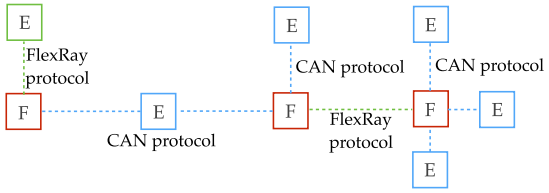
3.1. Abstracting the architecture of IVN systems

A practical IVN system contains several subnetworks following different protocols, and these communicate with other protocols through gateways, as shown in the upper part of Fig. 4. There are two kinds of subnetworks and buses, and gateways connecting different subnetworks in an IVN system. Within a subnetwork, there is also message transmission between nodes and such messages can be verified with a single protocol, but this work focuses on the communication between the subnetworks using heterogeneous protocols. Therefore, the first stage of abstraction is to simplify the architecture of IVN systems.

For abstracting the architecture of an IVN system, first, we need to find all gateways in the IVN system, and then separate the system from the gateways. There are subnetworks connected the gateways through a bus, or between gateways. These subnetworks are abstracted as corresponding environments, and then these environments are connected to gateways which the original subnetworks were connected by corresponding buses. Since the gateway is not the focus of our research, we consider forwarders instead of gateways for forwarding messages between environments. The buses are communication standards according to protocol specifications, therefore, we define bus protocols in the abstracted IVN system. We ignore the physical connections between nodes and this work is centered on communication rules based on protocol specifica-

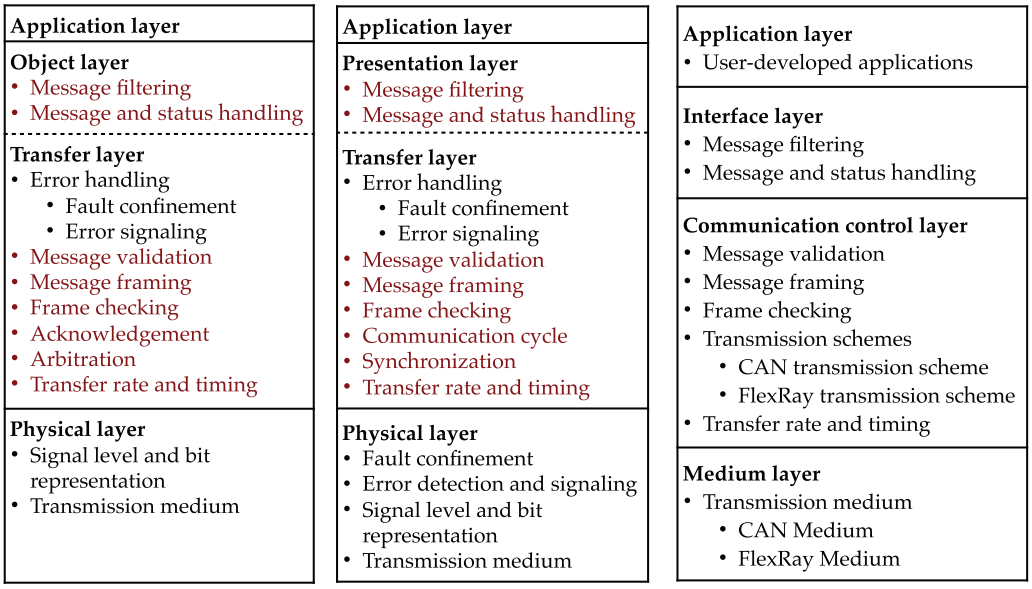


(a) The architecture of an IVN system.



(b) The abstracted architecture of an IVN system.

Fig. 4. Abstraction for the structure of IVN systems.



(a) CAN protocol. (b) FlexRay protocol. (c) A common structure.

Fig. 5. Layered structures of CAN and FlexRay protocols.

tions. By the abstraction, the architecture of the IVN system is shown at the bottom of Fig. 4 (b). The squares with *E* are CAN environments and FlexRay environments. They are connected to the corresponding forwarders as the above gateways. The dotted lines represent CAN protocol and FlexRay protocol.

An abstracted IVN system will consist of CAN environments, FlexRay environments, Forwarders, CAN protocol and FlexRay protocol. Although we have simplified the system structure and reduced the number of nodes, the communication standards in the system are still complex. So we need the next stage of abstracting communication protocols.

3.2. Abstracting the composition of protocol specifications

The abstract result of an IVN system has CAN environments and FlexRay environments. Although these environments are the abstraction of subsystems, their communication behaviors still need to conform to protocol specifications. The CAN specification and FlexRay specification respectively describe the communication behaviors in detail (Altran Tecnologies). However, a lot of details will result in an enormous number of states for the system, and make it difficult to check. Whereas if we simplify the system too much, it is hard to availably verify properties of the system. Therefore, a proper abstraction of the specifications is significant for checking IVN systems. Fig. 5(a) and (b) show the layered structure of the protocols.

Fig. 5 (a) shows the layered structure of CAN and the functions of each layer. The *Application layer* involves tasks run on an ECU and tightly depends on the actual IVN system being designed. The *Object layer* and the *Transfer layer* contain all data link services and functions. The principal function of the *Object layer* is to filter messages and handle the message status. This layer determines which frames are received and which frames are to be transmitted, and provides an interface between the *Application layer* and the *Transfer layer*. The *Transfer layer* represents the kernel of the CAN protocol, performing media access control (MAC). It controls frame transmission and ensures that frames are transmitted correctly. Within the *Transfer layer*, an error handling module detects errors in the bit levels and handles them appropriately, such as by shutting down defective nodes and retransmitting frames. Besides, the *Transfer layer* checks and distinguishes frames, then determines which frame is to be sent through the arbitration process. The *Physical layer* defines the low-level behavior related to how signals are transmitted in the transmission medium.

Fig. 5 (b) shows the layered structure of FlexRay nodes and lists the functions of each layer. The *Application layer* contains tasks developed by the application developers. The *Presentation layer* is similar to the *Objective layer* of CAN. It contains the communication controller host interface that connects the *Application layer* with the *Transfer layer*. The *Transfer layer* describes the kernel of FlexRay, defining the communication cycle, and handles timing and synchronization. Other functions are similar to those in the *Transfer layer* of CAN. The *Physical Layer* not only covers the actual transfer of bits between the different nodes but also detects errors in the time domain.

The specifications contain a lot of information about how to implement the protocols on hardware. Since the purpose of our research is to verify the IVN system in the design phase, behaviors associated with the physical layer are not necessary. Therefore, we abstract the protocol specifications to remove hardware related contents in each layer and ignore the *Physical layer*. The underlying implementation of the system is not involved in the system design model. On the other hand, the presented example in Section 2.3 shows that the timed property of the message transmission between CAN and FlexRay subsystems is affected by different communication mechanisms, even though no errors occur in the system. Therefore, this work focuses on the communication mechanism of CAN and FlexRay without error detection and error handling. We do not consider such functions in the *Transfer layer*. In Fig. 5 (a) and (b), we have highlighted the functions we care about in black font, leaving out the functions in gray font.

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

protocol that consists of the *Interface layer, Communication control layer* and *Physical layer* as shown in Fig. 5 (c). The *Application layer* is not defined in the specifications. It is used to implement data processing and control functions of IVN systems designed by users. However, the *Application layer* is an indispensable part of the IVN system. It is reserved in the common structure and used to generate messages that need to be transmitted. The object layer and presentation layer of the specifications are abstracted as the *Interface layer*. It connects the *Application layer* and *Transfer layer*, and is important for exchanging data between environments. The *Transfer layer* keeps communication schemes defined by the specifications, including message validation, frame checking, synchronization, communication cycle, arbitration, and transfer rate and timing for nodes of different protocols. Though we ignore all functions in the *Physical layer*, it still needs a transmission bus to represent the transmission of messages over hardware. So we abstract the *Physical layer* as *Medium layer* to store the frame transmitted on hardware. The functions of each layer are abstracted in the reusable framework proposed in the next section.

4. Framework

The diversity of IVN systems means that a reusable framework is necessary to model and verify the communication behavior between CAN and FlexRay. In this section, we present an overview of the proposed framework and describe how to model IVN systems using the framework.

4.1. Overview

Abstracted IVN systems may employ different numbers of environments, these environments may use different types of buses to connect gateways in various topologies. The diversity of IVN systems means that we need a reusable framework to model and verify communication behaviors with both CAN and FlexRay protocols. Referring to the protocol structure we presented in the previous section, we propose a reusable framework, including the *UPPAAL* engine, *Transmission medium, Communication controller, Interface, Environment, Forwarder* and *Configuration* modules, as shown in Fig. 6.

The *UPPAAL* is the foundation on which the other parts are built. The *Medium* module simulates message transmission over physical wires as the *Physical layer* of the specifications. The *Communication controller* contains CAN and FlexRay protocol models describing the communication mechanisms through the transfer layer of the nodes. The *Interface* contains several buffers designated for specific data streams called frames and operates as a communication bridge between the *Application layer* and *Communication controller*. The *Environment* module is task models that write/read message to/from the *Interface*, as the *Application layer* of the specifications. The *Forwarder* module also belongs to the *Application layer* and is a special task to communicate between different protocols. It executes frame-forwarding processes, representing an essential component to convert protocols. Finally, the *Configuration* module contains all parameters related to the other modules according to the system design.

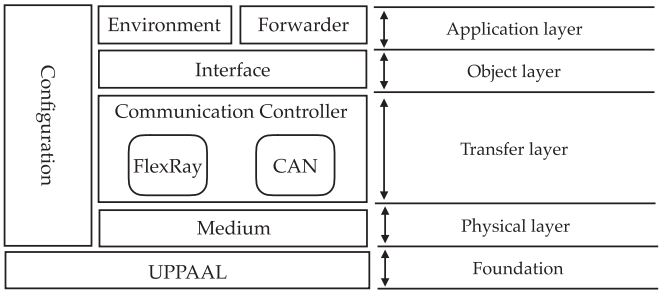


Fig. 6. Framework.

4.2. Modeling IVN systems

The framework can be used to construct a design model of an IVN system. We use a class diagram of the framework to display the relationships between modules, and all attributes and operations, as shown in Fig. 7. To reflect the feature of the modules, we define two special stereotypes, $<<Changeable>>$ and $<<Fixed>>$. The $<<Changeable>>$ means that the class can be changed according to the system design. The $<<Fixed>>$ means that the class is fixed and reusable in terms of modeling different IVN systems. In addition, some modules are created in the form of automata in UPPAAL. We draw a component diagram against Fig. 7 to show relationship between automata, as shown in Fig. 8. Combined with these two diagrams, we will describe in detail how to model IVN systems using the framework.

4.2.1. Communication controller module

The *Communication Controller* is the kernel for transmitting messages and performs communication behaviors following specifications. We separately define the *CAN* and *FlexRay* classes as $<<Fixed>>$, because they have their own special communication mechanism. We build fixed *CAN* and *FlexRay* models in UPPAAL, and these can be reused to construct different IVN systems. The *CAN communication controller* transfers messages based on the static priority scheduling algorithm. The *Arbitration* monitors the sending `CANBuffer[]` in real-time, and reads a message with the highest priority. The *Transceiver* is responsible for delivering the message to *CANMedium*, and stores the message to the corresponding receiving *CANInterface* when the transmission ends.

- The *CAN* class has two attributes, `CANBusID` and `index`. The `CANBusID` is a unique identifier for a bus. The `index` is a variable in automaton *Arbitration*. The *CAN* model is composed of two automata, *Arbitration* and *Transceiver*, which is built with reference to the CAN model in L.Waszniowskiâs work (Waszniowski et al., 2009). Unlike his CAN model, we simplify the *Transceiver* to perform transmissions of all message in a CAN subnetwork, as shown in Figs. 9 and 10.
  - The *Arbitration* monitors whether a CAN task send a transmission request in real-time. If the *Arbitration* supervises a request, a synchronization channel `transmissionRequest`, it will scan sending buffers of the CAN *Interface* from the highest priority identifier (`index=1`) to the lowest message priority (`index=maxPriority`). A message with higher priority wins the arbitration process. The arbitration result is synchronized to the *Transceiver* by the channel `arbSuccess`. The *Transceiver* automaton leaves the location `waitForTrans` to the location `waitForEnd`, as shown in Fig. 10. The *Arbitration* waits in the location `msgTrans` for the message transmission time given by a value *CANMsgLength*. When the message transmission is completed, the *Transceiver* synchronizes with the *Arbitration* using the channel `transEnd`. The *Transceiver* stores the message to the receiving buffer of the CAN *Interface* and resets the sending buffer `CANMsgForSend[index]`

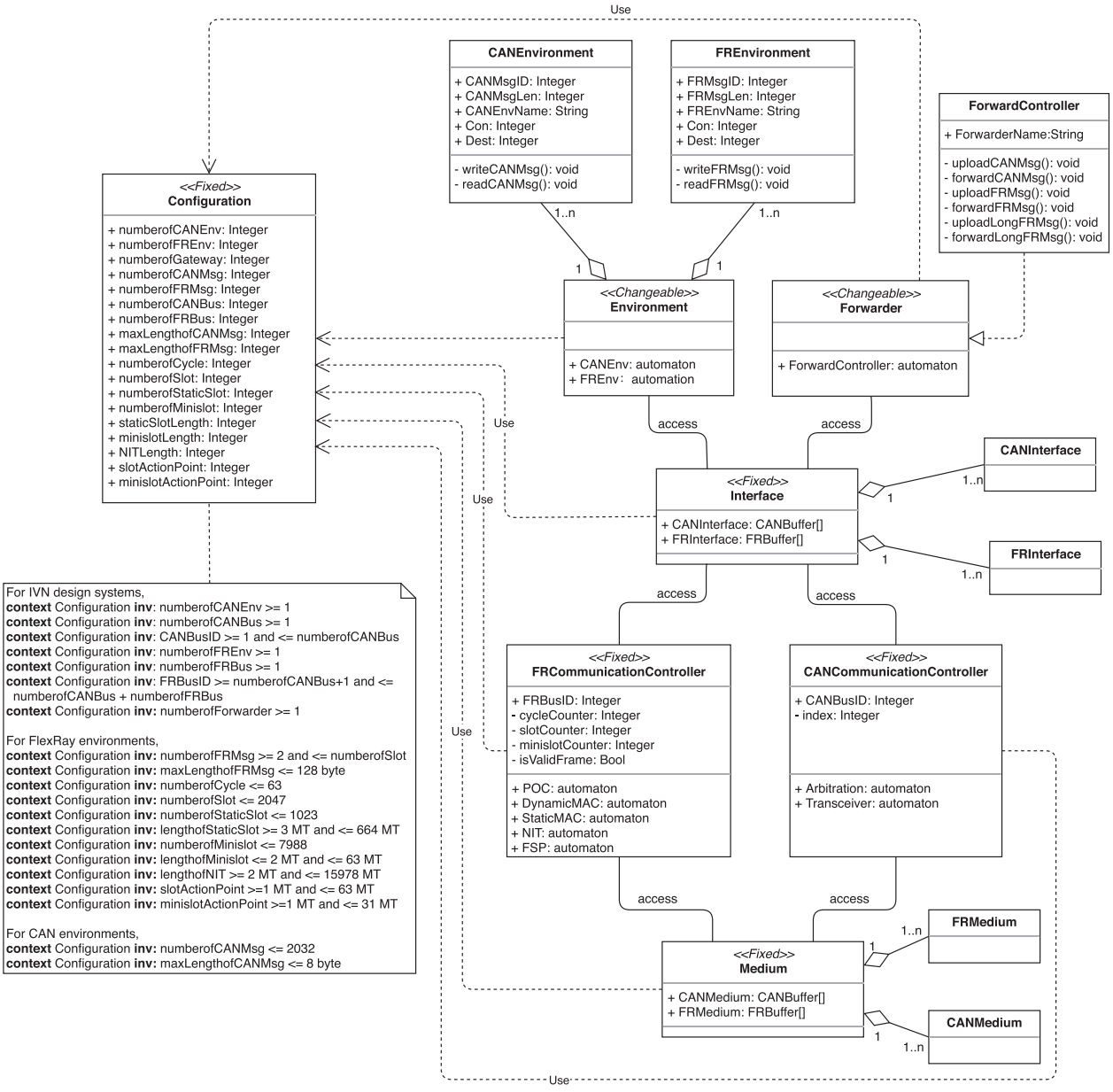


Fig. 7. Class diagram of the framework in UML.

by the function `receive()`, and clears the CAN bus by `resetBusCAN()`. Then the *Arbitration* will restart arbitration from the highest priority identifier. The channel `forward` is used to synchronize with a gateway model or a receiver model.

- The *FlexRay* class also has some attributes. The `FRBusID` is a unique identifier for a bus. The `cycleCounter`, `slotCounter` and `minislotCounter` are used to *FlexRay* model for counting the number of communication cycles, slots and minislots. The `isValidFrame` is a bool to show whether a frame is valid or not. The *FlexRay* model transmits messages in determinable time slots of communication cycle, which has several automata to serve communication cycles refer to the FlexRay specification (Altran Tecnologies). The protocol operation controller (*POC*) monitors the beginning and ending of a communication cycle, and counts the number of communication cycles. The *StaticMAC*, *DynamicMAC*, and network idle time (*NIT*) simulate the communication cycle and arbitrate messages. The frame and symbol processing (*FSP*) monitors the bus state and transmit or receive messages from the *FRInterface*. These automata synchronize through four channels, `cycleStart`, `DynamicStart`, `NITStart` and `cycleEnd`, as shown in Fig. 8. During the communication cycle, the *StaticMAC* and *DynamicMAC* inspect whether any messages need to be transmitted in the *FRInterface* at the beginning of each time slot. If there is, they will change the state of the FlexRay bus, and the *FSP* plays the *Transceiver* of the *CAN* model to access the *FRInterface* and transmits it to the *FRMedium*. When the transmission finished, the message will be stored in the *FRInterface* again by the *FSP*, and waits to be received or forwarded. If no messages need to be sent, they will wait for next time slot. We present all automata of the *FlexRay* model to illustrate how the protocol models transmitted messages (as shown in Figs. 11–15).
  - Under normal operation of a *FlexRay* system, the *POC* (as shown in Fig. 11) is going to start communication cycle,

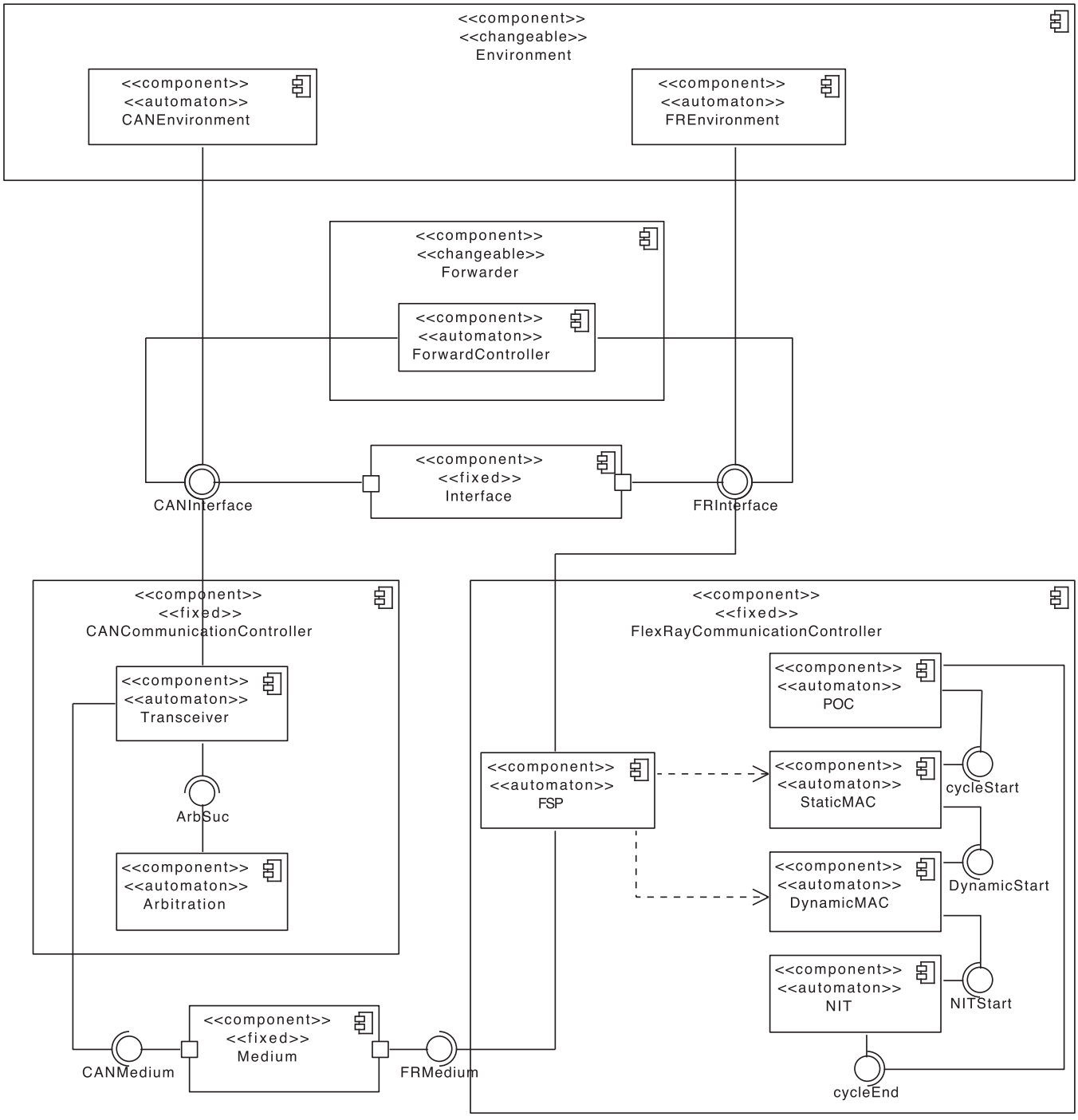


Fig. 8. The component diagram.

which immediately leaves the initial location `startUp`. The *POC* checks whether the value of the cycle counter is less than or equal to the maximum number of cycles. If the constraint is satisfied, the *POC* uses the channel `cycleStart` to *StaticMAC* and start a communication cycle. Then, the *POC* waits for the end of the cycle in the location `waitForCycleEnd` until it detects a synchronization channel `cycleEnd`. It will recheck the value of the cycle counter. If the value is less than the maximum value of the cycle counter, the *POC* will update the value plus one and reset the clock `x` for starting the next communication cycle. If the value is equal to `maxCycleCounter`, the *POC* will return to the initial state, and reset the clock.

- When the *POC* starts a communication cycle, it first enters the static segment with the `cycleStart` channel. In the static segment, there are some static slots with the same length and an action point. The *StaticMAC* automaton (as shown in Fig. 12) operates on each static slot repeatedly until the end of the static segment. For each static slot, after the action point, if there is a message waiting for transmission, the *StaticMAC* checks the validity of the message and

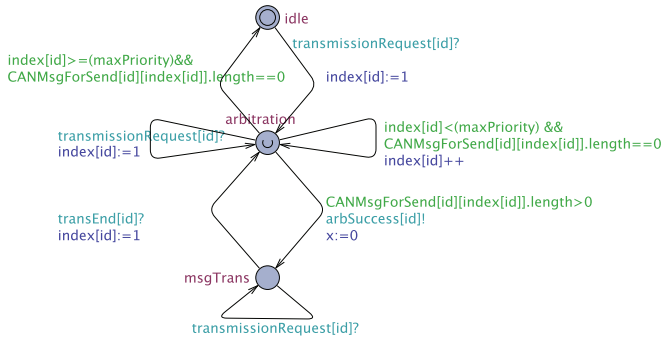


Fig. 9. Arbitration automaton.

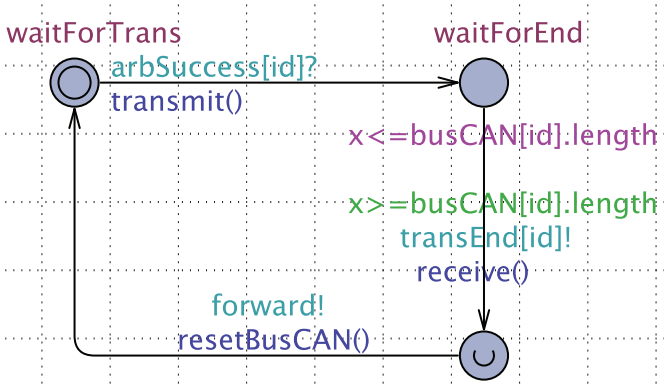


Fig. 10. Transceiver automaton.

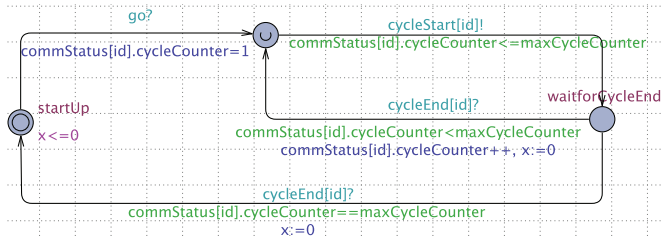


Fig. 11. POC automaton.

transmits it as a frame in that slot with the same identifier as the message. If no messages are waiting for transmission, the *StaticMAC* goes to another location to wait for the boundary of this slot. When the `slotCounter` reaches the maximum number of static slots, the communication cycle moves to the dynamic segment or goes to the *NIT* segment.

- After the *StaticMAC*, the *DynamicMAC* automaton will start with a channel, `dynamicStart`, as shown in Fig. 13, if the dynamic segment was defined. The *DynamicMAC* has two counters, the `slotCounter` keeps counting from the `maxStaticSlots+1`, and the `mimislotCounter` counts the number of minislots in the dynamic segment. Each slot in the dynamic segment also has an action point that is set to the larger value of the slot action point and the minislot action point. Before transmitting, the *DynamicMAC* also checks whether there is a message waiting to be transmitted in this dynamic slot and whether there are enough minislots for this message. If the constraint is satisfied, the bus status will be changed to `TRANS`, and the *DynamicMAC* transmits the message to the bus and moves to the location `waitforTransEnd`. When the transmission is complete, The *DynamicMAC* will recalculate the number of minislots and update the minislot counter according to the value of the current clock `x`. At the same time, the bus status is updated to `CHIRP`. When the clock `x` satisfies this dynamic slot boundary the dynamic slot ends and the bus becomes idle status (`busStatus=IDLE`). If no messages is waiting to be transmitted in this dynamic slot, or there are no enough minislots for this message, the length of the dynamic slot is a minislot. While a dynamic slot is finished, the *DynamicMAC* examines the values of the slot counter and minislot counter. If the value of `slotCounter` is less than the maximum number of slots, the *DynamicMAC* will perform the next dynamic slot from setting the dynamic action point. But there are no more mini-slots for frame transmission, that is the value of the minislot counter is `maxMinislotCounter`, the *DynamicMAC* will starts *NIT* segment and reset the minislot counter. If the value of `slotCounter` is equal to the `maxSlotCounter` and the minislots have just been used up, the dynamic segment ends and the *MAC* performs the *NIT* segment. If there are still some minislots, *DynamicMAC* will consume these minislots and then starts the *NIT* segment.
- The *NIT* (as shown in Fig. 14) is started with a channel `NITStart` from *DynamicMAC*, while the value of the slot counter is reseted to 0. Since the *DynamicMAC* already spent the remaining minislots in the communication cycle, the *NIT* only keeps the minimum length of the NIT segment. When the *NIT* ends (`x<=NITLength`), the *NIT* will communicate with the *POC* using the channel `cycleEnd`, In other words, the current communication cycle ends and the *POC* turns into next communication cycle.

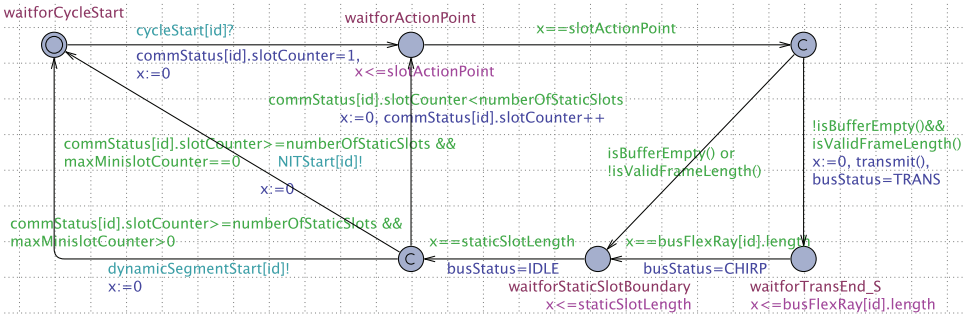


Fig. 12. StaticMAC automaton.

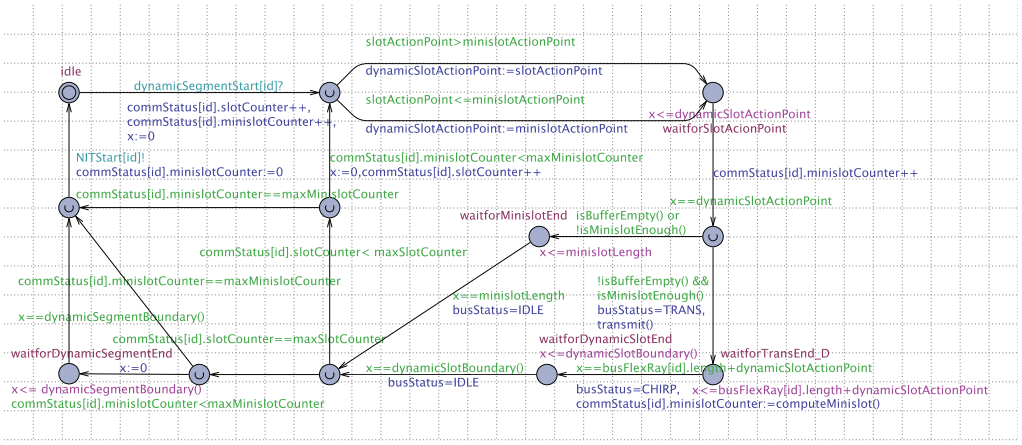


Fig. 13. Dynamic MAC automaton.

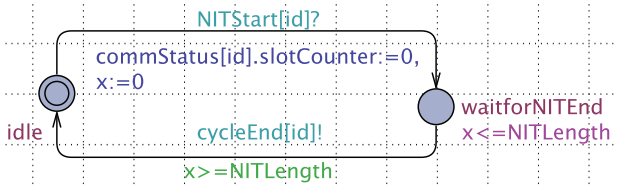


Fig. 14. NIT automaton.

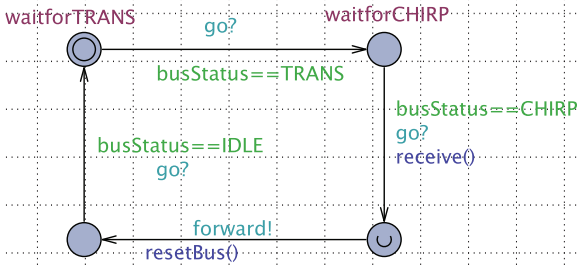


Fig. 15. FSP automaton.

- The *FSP* (as shown in Fig. 15) is waiting for a transmission. When the bus status becomes `TRANS`, there is a message transmitting. Then, the *FSP* is waiting for the bus status changed to `CHIRP`. When the message transmission finished, the *FSP* executes the function `receive()` to store the message to the FlexRay interface and checks whether the identifier of the message is matched with the current slot number. The *FSP* will notify receivers that the message has been received by channel `forward`, and reset FlexRay bus. When the current slot ends (`busStatus==IDLE`), the *FSP* turns into the initial location and waits next message transmission.

There are three functions related to transmission in both *Transceiver* and *FSP*. We show the functions for *FSP* of FlexRay in the following code fragment. The `transmit()` gets a message from a sending buffer of the *Interface* to the *Medium* and empties the data of the buffer. The `receive()` receives a message from the *Medium* and store it in a receiving buffer of the *Interface*. The `resettBus()` resets the *Medium* to idle. The *Transceiver* of CAN has similar functions to the *FSP*.

```
void transmit(){
    FRMedium.id = commStatus.slotCounter;
    FRMedium.length = FRMsgForSend[commStatus.slotCounter].
    length;
    FRMedium.envir = FRMsgForSend[commStatus.slotCounter].
    envir;
    FRMedium.dest = FRMsgForSend[commStatus.slotCounter].
    dest;
    FRMsgForSend[commStatus.slotCounter].id = 0;
    FRMsgForSend[commStatus.slotCounter].length = 0;
    FRMsgForSend[commStatus.slotCounter].envir = 0;
    FRMsgForSend[commStatus.slotCounter].dest = 0;}

void receive(){
   FRMsgForReceive[FRMedium.id].id = FRMedium.id;
   FRMsgForReceive[FRMedium.id].length = FRMedium.length;
   FRMsgForReceive[FRMedium.id].envir = FRMedium.envir;
   FRMsgForReceive[FRMedium.id].dest = FRMedium.dest;}

void resetBus(){
    FRMedium.id = 0;
    FRMedium.length = 0;
    FRMedium.envir = 0;
    FRMedium.dest = 0;    }
```

4.2.2. Interface and medium modules

Interface module is defined by a *Interface* class with *<<Fixed>>*, and aggregates the *CANInterface* class and *FRInterface* class. The *Interface* provides buffers for storing messages exchanged between the *Application layer* and *Transfer layer*. The *Medium* class defines two kinds of hardware wires, *CANMedium* class and *FRMedium* class. They are represented by two buffers that correspond to frame structures. Although CAN messages and FlexRay messages have different frame formats, not all data fields of the frame are necessary in our model, for example, the cyclic redundancy check (CRC) field is used to guarantee the correctness of the frame. Because our model is an abstraction for an error-free communication system, the data field concerning error checking is ignored. An abstracted frame format is proposed for both CAN

messages and FlexRay messages, including a frame identifier, payload length, an environment identifier, and a destination identifier.

The following code fragment shows frame structures for interfaces and bus mediums in UPPAAL. These buffers use the same data structure defined by protocol frames. Each buffer includes four information, the length, and identifier of messages (`id` and `length`), the identifier of a bus connected to an environment that sent the message (`envir`) and the identifier of a bus connected to an environment that receives the message (`dest`). We set up two kinds of buffers for CAN and FlexRay, sending buffers (`CANMsgForSend[]` and `FRMsgForSend[]`) and receiving buffers (`CANMsgForReceive[]` and `FRMsgForReceive[]`). The sending buffers are used to store messages that need to be sent, and the receiving buffers are used to receive messages that have completed transmission. The number of the CAN buffers are defined by the sum number of CAN messages and FlexRay messages. The *CANMedium* and *FRMedium* are two kinds of buffers with the frame structure for simulating physical wire.

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

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

4.2.3. Environment module

The *Environment* class is presented by a *<<Changeable>>* stereotype. Generally, each node may has different tasks to perform various operations and processing on data in the IVN system. Although we can ignore the concrete functionalities of applications in ECUs, the possible tasks are still infinite. Thus, the *Environment* model could be changed by designers. The *Environmnet* has `CANEnv` and `FREnv` automata built in UPPAAL, because we need two types of messages to be sent from the CAN and FlexRay environments, respectively. These automata execute write-read operation to access the *Interface*. We take FlexRay environments for instance. The function `writeFRMsg()` writes a message into a sending buffer of the *FRInterface* corresponding `id`, and the function `readFRMsg()` reads messages from the receiving buffer of the FlexRay interface and clears data in the buffer, as shown in the following code fragment. Similarly, there are two functions, `writeCANMsg()` and `readCANMsg()` are used in *CANEnv* automata and access *CANInterface*.

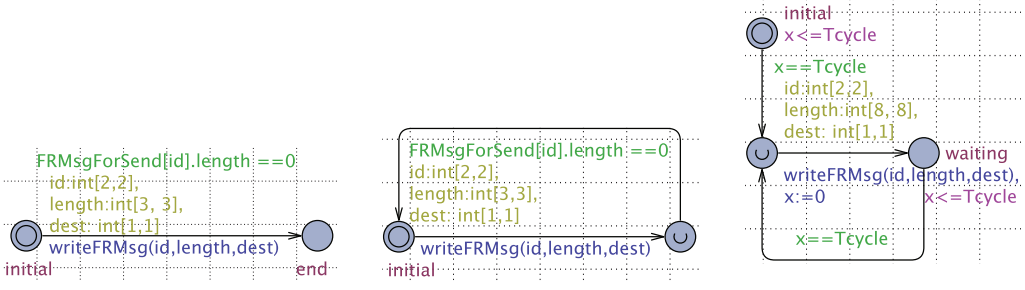
```
void writeFRMsg(FRMsgID id, FRMsgLength length,
     BusID envir, BusID dest){
     FRMsgForSend[id].id = id;
     FRMsgForSend[id].length = length;
     FRMsgForSend[id].envir = envir;
     FRMsgForSend[id].dest = dest;}

void readFRMsg(FRMsgID id){
     FRMsgForReceive[id].id = 0;
     FRMsgForReceive[id].length = 0;
     FRMsgForReceive[id].envir = 0;
     FRMsgForReceive[id].dest = 0;}
```

In addition, the `CANEnv` and `FREnv` may have different time descriptions, and we focus on checking the timed properties of the IVN system. Hence, we present three kinds of task automata for modeling periodic tasks, cyclic tasks, and sporadic tasks in UPPAAL, as shown in Fig. 16. The periodic task writes a message to the *Interface* based on a time period (`x==Tcycle`). The cyclic task writes a message when the specific buffer is empty (`FRMsgForSend[id].length==0`). The sporadic task is executed only once, and writes a message to the *Interface* using the function `writeFRMsg()`. The possible tasks are not limited to these three types and can be changed by the user.

4.2.4. Forwarder module

Gateways are critical in bus communication with several protocols. The gateways provide multiple operations between networks, such as protocol translation, routing, and data exchange. The efficiency of data transmission between two networks is directly affected by the gateway that connects them. We can improve the mapping and scheduling to enhance the performance of commu-



(a) The sporadic task (b) The cyclic task (c) The periodic task

Fig. 16. Task automata.

nication between CAN and FlexRay, such as (Schmidt et al., 2010; Kim et al., 2015). These functions belong to the application layer of the ISO/OSI model, like some tasks running on an ECU of a node, and the application layer can be varied because CAN and FlexRay specifications do not describe the application layer. Our work is to check message transmission by abstracting the communication mechanisms of the protocols in the system design phase. Messages are from the application layer and transmitted by the protocols. The protocols are specified and fixed, but the application layer is diverse and can be designed and changed. Therefore, we provide the framework to check changeable system designs, including the topology and the application layer both of the tasks of nodes and gateways. Therefore, we only implemented the basic functions of the gateway as a forwarder for evaluating our framework.

The *Forwarder* is also a *<<Changeable>>* class, and can be realized by a *ForwardController* automaton depending on the IVN system design. It is to forward messages between the *CAN* and *FlexRay*, as a gateway. In reality, gateways have different features, such as scheduling algorithms, which may affect the time property of the IVN system. Thus, the *Forwarder* module can be changed based on designs. The process of forwarding messages is executed by two functions for each protocol. The uploading functions, `uploadCANMsg()` and `uploadFRMsg()`, read a CAN/FlexRay frame from the *Interface* and upload the frame in a temporary buffer. The forwarding functions, `forwardCANMsg()` and `forwardFRMsg()`, convert the format of the CAN/FlexRay frame and forward the frame to a corresponding buffer in the *Interface* as a FlexRay/CAN frame. Since the length of the FlexRay message may be longer than the maximum length allowed for CAN messages, such messages will be split into multiple CAN frames with the same identifier using the `uploadLongFRMsg()` function and these CAN frames will be forwarded to a CAN environment one by one using the `forwardLongFRMsg()` function.

We implement a `ForwardController` automaton with some essential functions based on an existing work (Zhao et al., 2010), such as monitoring messages, converting protocols, and forwarding messages. The `ForwardController` monitors the *Interface* in real-time. When a message needs to be forwarded, the automaton leaves the location `idle`, and goes to the urgent location `monitor_interface`, as shown in Fig. 17. The three transitions from the location indicate the scheduling policy of the forwarder: 1) if there are only FlexRay messages in the *Interface*, the automaton forwards a FlexRay message with the smallest identifier number to a CAN environment; 2) if there are only CAN messages in the *Interface*, the automaton forwards a FlexRay message of the highest priority to a FlexRay environment; 3) if there are both FlexRay and CAN messages in the *Interface*, the forward controller will deal with a CAN message first and then a FlexRay message. At last, when there are no messages in the *Interface*, the automaton returns to the location `idle`.

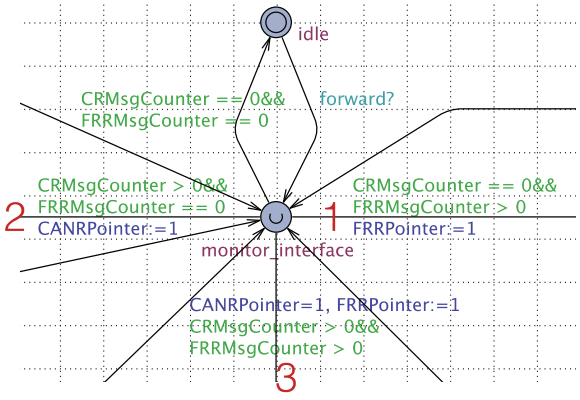


Fig. 17. *ForwardController* fragment for monitoring the *Interface*.

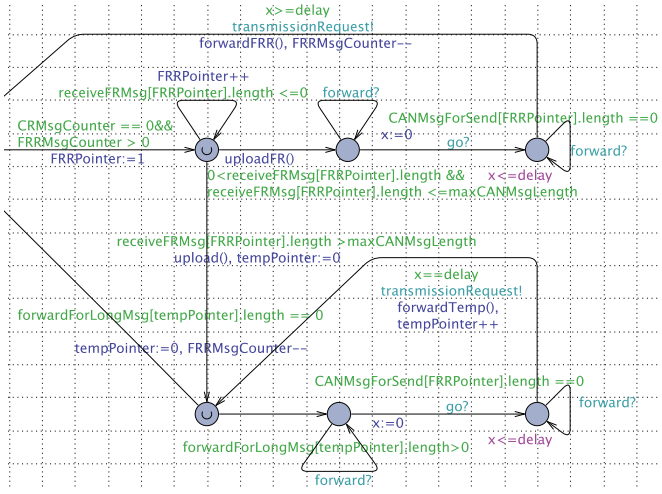


Fig. 18. *ForwardController* fragment for forwarding messages.

Taking the first transition as an example, Fig. 18 is used to introduce the process of forwarding messages. When a FlexRay message arrives in its receiving buffer first, the constraint, `CRMsgCounter==0` and `FRRMsgCounter>0` are satisfied. The gateway updates `FRRPointer` to 1, that is the gateway will check the receiving buffers of FlexRay *Interface* from the buffer with the smallest identifier. If the first buffer is no message, `FRRPointer` will plus 1 to inspect the next buffer until a buffer has a FlexRay message. If a message was found, the gateway checks whether the message needs to forward to the other environment, by comparing the value of `envir` and `dest`. If their values are the same, the gateway will look for the next message. Before forwarding the message, the length of the message will be checked by a constraint. If the length meets the configuration of CAN message, this message is directly copied to a temporary buffer by the function `uploadFR()` and waits for forwarding. The gateway checks the sending buffer of CAN *Interface* corresponding to the message. If the sending buffer is empty, the gateway transfers the message to the sending buffer by function `forwardFRR()`, and sends a signal `transmissionRequest` to CAN model. Here we consider a delay for forwarding messages in the gateway. After forwarding a message, the gateway goes back to the urgent location `monitor_interface`. If the length of the FlexRay message is beyond the configuration of CAN message, this message will be divided into several messages with the maximum length of CAN message and the same configuration by the function `upload()`. These messages are stored in temporary buffers, and will be forwarded one by one to the same sending buffer of CAN `Interface` according to the identifier when the sending buffer is empty. As all messages in the temporary buffers are forwarded, the gateway automaton is back to the urgent location `monitor_interface`.

4.2.5. Configuration

The framework can be reused for modeling different IVN systems by modifying various parameters. The *Configuration* class holds all parameters in the *Environmnet, Forwarder, Interface, CAN, FlexRay* and *Medium*. We describe them and give their range of values, as shown in the bottom left of Fig. 7. To build an IVN system model, we need to specify the number of CAN and FlexRay environments and the forwarders and then set up each environment. For CAN environments, we only need to define the identifier and length of each message. For FlexRay environments, we also need

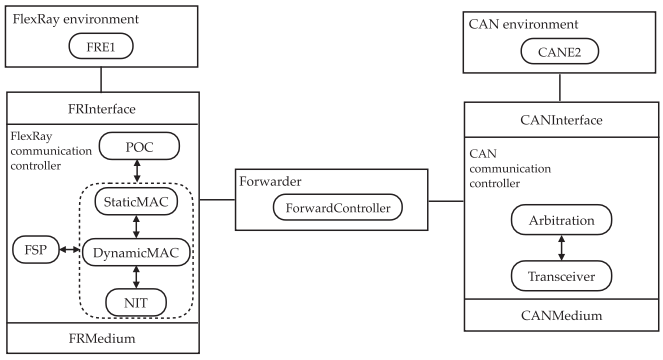


Fig. 19. Architecture of the IVN system design.

Table 1
Parameters of the IVN system design.

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

to configure the communication cycles. These parameters are described using the object constraint language following the protocol specifications and constraints. We use MT (the smallest time unit in the communication cycle) to represent the execution time of messages on the bus.

5. Experiments

The framework is evaluated in terms of four aspects. Firstly, a basic but non-trivial IVN system design is used to evaluate the validity of the framework by checking a series of essential properties with UPPAAL. Secondly, we check the reachability and response time of messages in the IVN system. Thirdly, the applicability of the framework is illustrated by designing three typical IVN system topologies and comparing their timed properties; the reusability of the framework is evaluated by matching the source code of different system models in UPPAAL. The performance of the framework is tested by a design model for IVN systems within the scope of message transmissions.[1]

5.1. An IVN system design

We consider an IVN system design for evaluating the validity of the framework and analyzing the timed properties of IVN systems. This IVN system has two environments, a FlexRay environment (E1) and a CAN environment (E2), which are connected by a forwarder (see Fig. 19). Each layer of the environments and gateways comes from the framework described in Section 4. The parameter settings in *Configuration* are listed in Table 1. There are two cyclic tasks, one in E1 and one in E2, and these are the *Environment* of the framework. A FlexRay message `msg1` (`id=2`, `length=3`) will be sent to E2 and a CAN message `msg2` (`id=1`, `length=2`) will be sent to E1.

[1] UPPAAL models used in this paper: https://github.com/xiaoyun-guo/IVN-UPPAAL.

5.2. Validity

In the framework, the protocol module is vitally critical for transmitting messages. We implement the system design model for the system design described above. This system design has all types of models in the *Application layer* and two kinds of messages, which are sent from different environments. Thus, it is relatively simple, but non-trivial, and so can be used to evaluate the validity of the framework. We check some properties to guarantee the correctness of communication between the CAN and FlexRay model. These properties describe both states and paths of the model to represent the communication mechanisms, which are defined by the CAN and FlexRay protocol specifications. We give the precise expressions of the properties by a specification language, timed computation tree logic (TCTL), as queries in UPPAAL model checker (Behrmann et al., 2004). All the checking of properties has only two results given by UPPAAL, "satisfied" or "unsatisfied". We list the properties as follows.

- **Check1:** Is the system model deadlock free?
  `Query1: A[] not deadlock`
  This query is satisfied. The system model will never be deadlock. The deadlock freeness is an important property for verifying the credibility of the model.
- **Check2:** Are the messages transmitted and received in the assigned slots in FlexRay?
  `Query2: A[] forall(i:int[1,3])(FRMsgForReceive[i].length > 0) imply (commStatus.slotCounter==i)`
  This query is satisfied. In this model, we set up three slots in the FlexRay communication cycle. All messages are correctly received in the allocated slots. The `msg1` will be received in slot 2, and the `msg2` will be received in slot 3. Thus, the FlexRay model can control the message sending and receiving processes following the slot number.
- **Check3:** Each message transmission monopolizes the bus. Does only one buffer receive the transmitted message in a slot?
  `Query3: A[] forall(i:int[1,3]) forall (j:int[1,3]) (FRMsgForReceive[i].length>0)&& (FRMsgForReceive[j].length>0) imply (j=i)`
  `Query4: A[] forall(i:int[1,2]) forall (j:int[1,2]) (CANMsgForReceive[i].length>0)&& (CANMsgForReceive[j].length>0) imply (j=i)`
  The two queries are satisfied. It means that only one message will be received by a buffer of FlexRay or CAN. If two receiving buffers in *Interface* get messages at the same time, the identifier of the messages should be the same. That is only one message utilizes the bus in a moment of the system.
- **Check4:** Does the application model output messages successfully?
  `Query5: E<> forall(i:int [1,3])(FRMsgForSend[i].length>0)`
  `Query6: E<> forall(i:int [1,2])(CANMsgForSend[i].length>0)`
  The two queries are satisfied. The application model can write messages to *Interface*.
- **Check5:** Will the message in the sending buffers be sent?
  `Query7: forall(i:int [1,3])((FRMsgForSend[i].length>0) imply (FRMsgForSend[i].length==0)`
  `Query8: forall(i:int [1,2])((CANMsgForSend[i].length>0) imply (CANMsgForSend[i].length==0)`
  The two queries are satisfied. The CAN and FlexRay models can finish transmissions for all messages.

We examined the `Check1-5` to ensure whether messages were stored in the interface correctly, that is, messages waiting in the sending buffers are transmitted by the CAN/FlexRay model and

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

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

- **Check6:** Can the msg1 be transmitted from *E1* to *E2*?
  Query9: E<> (FRMsgForSend[2].length > 0)
  Query10: (FRMsgForSend[2].length > 0)--> (FRMsgForSend[2].length == 0)
  Query11: (FRMsgForSend[2].length>0)-->(busFlexRay.id == 2 && busFlexRay.length > 0)
  Query12: (busFlexRay.id == 2 && busFR.length == 0)-->(FRMsgForReceive[2].length > 0)
  Query13: (FRMsgForReceive[2].length > 0)-->(CANMsgForSend[2].length > 0)
  Query14: ((CANMsgForSend[2].length > 0)-->(CANMsgForSend[2].length == 0)
  Query15: (CANMsgForSend[2].length > 0)-->(busCAN.id == 2 && busCAN.length > 0)
  Query16: (busCAN.id == 2 && busCAN.length == 0)-->(CANMsgForReceive[2].length > 0)
- **Check7:** Can the msg2 be transmitted from *E2* to *E1*?
  Query17: E<> (CANMsgForSend[1].length > 0)
  Query18: (CANMsgForSend[1].length > 0)-->(CANMsgForSend[1].length == 0)
  Query19: (CANMsgForSend[1].length>0)-->(busCAN.id == 1 && busCAN.length > 0)
  Query20: (busCAN.id == 1 && busCAN.length == 0)-->(CANMsgForReceive[1].length > 0)
  Query21: (CANMsgForReceive[1].length > 0)-->(FRMsgForSend[3].length > 0)
  Query22: (FRMsgForSend[3].length > 0)-->(FRMsgForSend[3].length == 0)
  Query23: FRMsgForSend[3].length>0)-->(busFlexRay.id == 3 && busFlexRay.length > 0)
  Query24: (busFlexRay.id == 3 && busFR.length == 0)-->(FRMsgForReceive[3].length > 0)

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

These properties include message arbitration, transmission mode, and transport correctness of transmissions, which exist in the fixed modules of the system model. The checking results indicate the fixed modules of the IVN system design model meet these properties, so the system model based on our framework is valid. Since these modules can be reused to other IVN system designs, the validity of this design system will also be satisfied in other system models. Therefore, we can conclude that the proposed framework is valid for normal message transmissions under the CAN and FlexRay protocols.

5.3. Reachability

Any message loss can cause serious security problems, even if the bus structure is simple. The bus structure in our work is an abstracted topology, which consists of two buses at least, a CAN bus and a FlexRay bus. Also, transmission mechanisms on the two kinds of buses will impact the reachability, as well as the system design, such as the cycle of tasks. Therefore, the reachability is necessary for a safety-critical system. To verifying the safety property of the IVN system, the reachability of messages is considered. When there are many messages transmitted between a CAN environment and a FlexRay environment, it is essential to verify whether all messages can be received or if there is a message lost. Here, we also use the IVN design model but change the application model. For the system with two messages that need to be sent repeatedly, we used two periodic tasks to send msg1 and msg2 in the system design model. The results of Query25 and Query26 show that msg1 and msg2 cannot always be received, which are not satisfied. As the priority of msg2 is lower than that of msg1, it cannot be sent in the CAN environment. Moreover, messages are frequently sending through the gateway, the gateway becomes congested. So some messages are lost during transmission.

Query25[Y]: (CANMsgForSend[1].length>0)-->(receiveFRMsg[3].length>0)

Query26[N]: (FRMsgForSend[2].length>0)-->(receiveCANMsg[2].length>0)

5.4. Response time

A major feature of the framework is the description of behaviors with timed constraints. The transmission time of messages can be checked using an observer model that monitors specific model states. A sender task can send messages periodically, while a destination task receives messages immediately when the interface detects the presence of data. The message is stored in a sending buffer until it is received from the receiving buffer by the destination node. This process includes the transmission time of the message on buses as well as the waiting time of the message in the interface. The time required for this process is called the response time. We can check a series of timed properties to determine the best-case response time (BCRT) and worst-case response time (WCRT) of frames. UPPAAL defines a clock that does not represent a real time. In our model, the time unit of the clock is a macrotick (MT), which is the smallest time unit in the communication cycle of FlexRay. For example, if this MT is set to 1 *us*, the time unit in the model is 1 *us*. In the system design model, msg1 is sent from the FlexRay environment *E1* and transmitted to the CAN environment *E2*. We check the response time of msg1 with an observer model. Query27–30 verify BCRT and WCRT for msg1.

Query27[N]: E<>(observer.E2_receive && observer.x < 34)

```
Query28[Y]: A[](observer.E2_receive imply
observer.x >= 34)
Query29[N]: E<>(observer.E2_receive &&
observer.x > 94)
Query30[Y]: A[](observer.E2_receive imply
observer.x <= 94)
```

In each query, [Y] or [N] indicates whether the checking result of the query is satisfied or not. The BCRT and WCRT values for a query are currently determined by trial and error based on the parameters of the IVN system, such as the message length, communication protocols, and gateway delay. The results of Query27 and Query28 indicate that the response time of msg1 is never less than 34, which means that msg1 has BCRT = 34. The results of Query29 and Query30 show that msg1 has WCRT = 94. We also considered msg1 to have a length of 9, which is above the maximum length of CAN messages, when sent from *E1* to *E2*. In this case, BCRT and WCRT were 100 and 160. The response time of a message is a time interval between BCRT and WCRT by exhaustively checking all states of the system model. Thus, the framework can exactly check the timed properties with the help of the observer model.

The system model represents the transmission of messages between two different protocols, without the physical layer and error handling. The response time checked based on the model is a theoretical value used to illustrate the satisfiability of the system design. For a reachable message, the response time may not satisfy the required value. The response time is affected by both the topology and other application tasks. If the worst-case response time of a message cannot meet the requirements of the system design based on our model, it is also difficult to meet in the real system. Next, we discuss the response time of a message under different system designs.

5.5. Applicability

IVN systems have varied and complicated topologies, in which subsystems with different protocols are connected by gateways. We considered the central, backbone and daisy chain topologies with the following gateways (see Fig. 20).

- The central topology has a single gateway in the center, and other environments connect through the gateway. Communication between environments must pass through the gateway. If there are many messages to be transmitted, the gateway may become congested, resulting in a response time delay.
- The backbone topology has multiple gateways connected by a bus following a single protocol. Each gateway is connected to an environment. Communication between environments needs to be forwarded through two gateways. The message response time is not affected by the number of environments but is related to the number of messages.
- The daisy chain topology consists of gateways and environments in an alternating chain. The response time is a function of the distance between environments. The number of messages also influences the response time, as many messages may cause network congestion.

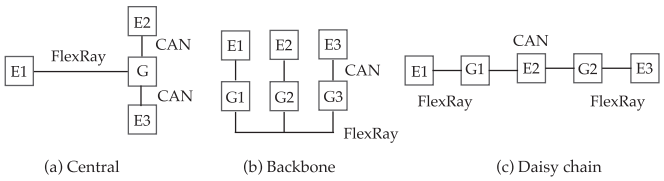


Fig. 20. Three topologies of IVN systems.

Table 2
Response time of each topology in the three cases.

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

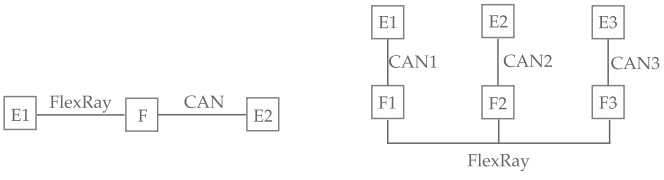
These topologies have unique characteristics, and the response time is affected by the number of environments and the number of messages. For each of the three IVN system topologies, we implemented three cases and compared their response time (see Table 2). Note that CAN, FlexRay, and the gateways and interfaces had the same parameter settings.

- In *Case 1*, two environments make up the three topologies.
  - The central and daisy chain topologies have the same structure, and msg1 (id=1, length=3) is sent from the FlexRay environment to the CAN environment. All response times are the same.
  - For the backbone topology, we consider two CAN environments connected by a FlexRay bus, so there are two gateways between the CAN environments. msg1 (id=1, length=3) needs to be forwarded twice to the other environment. The BCRT and WCRT values of msg1 are longer than for the other topologies.
- In *Case 2*, there are three environments are used to construct IVN systems on each topology, as show in Fig. 20 shows. msg1 (id=1, length=3) is sent from E1 to E3.
  - For the central and backbone topologies, even though there is one more environment than in *Case 1*, the transmission process of msg1 is unaffected. The BCRT and WCRT values in *Case 2* are the same as in *Case 1*.
  - The system model with daisy chain topology has a CAN environment between *E1* and *E3*, i.e., the number of gateways is greater than in *Case 1*. That is, msg1 needs to be forwarded twice to the destination environment *E3*, so the BCRT and WCRT values are greater than for *Case 1*.
- In *Case 3*, the structure is the same as for *Case 2*, but two messages are transmitted. msg1 (id=1, length=3) is sent from E1 to E3; msg2 (id=2, length=3) is sent from E2 to E3.
  - In the central and daisy chain topologies, msg2 may cause a forwarding delay in the gateways because the bus is busy. Therefore, the WCRT values of the central and daisy chain topologies will increase, and the BCRT values remain the same as in *Case 2*.
  - In the backbone topology, the response time is unchanged. Because messages with different identifiers are transmitted in the same communication cycle in the FlexRay network, the number of messages does not affect the response time.

Based on these results, we determined the impact of each topology on the message transmission process. When many environments are connected to the central gateway in the central topology, WCRT will increase. If there are many environments connected by the backbone topology, the response time will be affected by messages from other environments. When there are many environments between the sender and receiver in the daisy chain topology, the response time will increase. These results are consistent with the topological characteristics, and so we conclude that the framework can be reused to implement IVN system design models with different topologies.

Table 3
Comparison of source code of the three systems.

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



(a) System 1 with star topology.

(b) System 2 with backbone topology.

(c) System 3 with daisy chain topology.

Fig. 21. Three systems with different designs.

5.6. Reusability

We depicted the behaviors of each module in chapter 3 and chapter 4. In the framework, we defined the <<Fixed>> stereotype to describe *CAN, FlexRay, Interface, Medium* and *Configration* module. These modules can be reused to build different IVN systems with little modification. We consider three IVN systems with a different number of environments, buses, and topologies. These three systems have been introduced before, and their architectures are shown in Fig. 21. We will compare the models of the three systems, and discuss the parts of the model that can be reused and need to be changed.

The *System1* has two environments and a forwarder, which is the IVN system design model described in section 4.6. The *System2* has three environments and three forwarders withbackbone topology, which is the system with the backbone in the *Case3* in section 5.3.2. The *System3* has three environments and two forwarders with daisy chain topology, which is the system with the daisy chain in the *Case3* in section 5.3.2. We illustrate reusability by comparing the source codes of the three system models in UPPAAL. Table 3 shows the number of code lines in each system. First, we divide code of systems into four parts, *Configuration, CAN protocol, FlexRay protocol* and *Environments and forwarders*. Then, we calculate the amount of code in each part of the systems, and use the *System1* as a benchmark. After that, we compare the code of each part between the systems.

- The *Configuration* segment of the source code is the declaration part for defining global variables. All systems have 103 lines. There are 8 lines of code in the *System2* and *System3* that are different from the *System1*, accounting for 7.7% of the *Configuration*, because we set different numbers of environments, protocols, and forwarders.
- The *CAN protocol* segment has *CANInterface, Arbitration, Transceiver* and *CANMedium* that implement transmission behaviors and connection with environments and forwarders. In these three systems, they have the same code of the *CAN protocol*.
- The *FlexRay protocol* segment has *FRInterface, FRMedium* and five automata, *POC, StaticMAC, DynamicMAC, NIT* and *FSP*, which implement transmission behaviors and connection with environments and forwarders. These three system have the same code of the *FlexRay protocol*.
- The *Environments and forwarders* segments are changeable in the framework. All systems have different designs, such as different topologies and environment designs. Therefore, we did not compare the reusability of this segment of the code.

For different IVN systems, we only need to build the *Environment* module and *Forwarder* module based on the system design, and then change parameters of the*Configuration* for setting other modules.

5.7. Performance

To evaluate the performance of the framework, we use the three types of tasks, sporadic task, cyclic task, and periodic task, as our application models for comparison. The design system to which these tasks are applied has the same structure, which is a CAN environment and a FlexRay environment connected by a gateway. For each type of task system, we increase the number of message identifiers sent by the task, and verify three properties for each case as follows:

- *P1*: "The system model is deadlock free."
- *P2*: "The message (msg2) is always received."
- *P3*: "The response time of the message is greater than33.//

All properties are verified by UPPAAL 4.1.14 on a Mac OSX E1 Capitan machine with an Intel Core i7 3 GHz processor and 16 GB RAM. The memory usage and CPU times are listed in Table 4, where "out of mem." means more than 10Gb. When we use sporadic tasks to send messages, the three properties are satisfied for any cases. The performance of the framework is the best because each message is sent only once. When periodic tasks are set to the application model, the properties are also satisfied, but UPPAAL takes more time to verify them and memory than the sporadic tasks. The periodic tasks send messages with different periods so that there are a great many of possibilities for the transmission state of each message in the system. When design models use cyclic tasks as their application models, *P1* and *P3* are met, however, *P2* is not satisfied. Since cyclic tasks do not have a fixed time interval to send messages, the message may be overwritten. Besides, the state of the messages in the system can be infinite, so the state space of the system increases dramatically as the number of message identifiers.

Although we can only verify messages with few different identifiers, the number of messages can be huge. The environment is an abstracted subnetwork, and only performs tasks that require communication with other subnetworks. We defined a set of messages with a specific identifier to represent all messages that are sent from an environment to the other. Furthermore, messages with the same identifier are sent repeatedly and model checking

Table 4
Time and memory requirements of the verification by three types of tasks.

No. of message IDs	Property	Sporadic task		Periodic task		Cyclic task	
		Time(s)	Memory (Mb)	Time(s)	Memory (Mb)	Time(s)	Memory (Mb)
1	P1	0.018	8.953	0.220	9.027	0.255	15.125
	P2	0.012	10.363	0.305	11.328	0.149	18.449
	P3	0.08	8.922	0.733	10.910	0.640	13.164
2	P1	0.122	13.172	0.101	93.515	54.754	552.422
	P2	0.122	17.727	0.132	94.113	2.682	552.422
	P3	0.075	12.625	0.265	94.492	353.826	2,442.617
3	P1	0.636	58.289	139.807	417.093	–	out of mem.
	P2	0.654	58.762	431.332	612.000	0.497	304.523
	P3	1.050	60.563	481.741	1,465.613	–	out of mem.
4	P1	2.662	139.219	258.387	728.715		
	P2	2.553	146.398	745.271	1,105.973		
	P3	5.284	179.168	829.027	2,371.477		
5	P1	14.265	699.027	–	out of mem.		
	P2	13.147	723.684	–	out of mem.		
	P3	31.326	874.113	–	out of mem.		
6	P1	79.485	3,428.340				
	P2	73.308	3,548.090				
	P3	172.866	4,323.941				

exhaustively checks all situations where messages are sent with different delays. Therefore, the framework is effective for verifying message communications between several environments. If there are many environments and messages, the capability of the framework is limited.

6. Related work

6.1. Verification of IVN systems based on integration platforms

In the automobile industry, integration platforms are one solution for the IVN system design process (Sangiovanni-Vincentelli, 2003; Demmeler and Giusto, 2001). For example, Daimler-Chrysler laboratory developed a tool including software and hardware architectures to flexibly construct and test IVN systems (Grimm, 2003); T. Demmeler et al. proposed a virtual integration platform to simulate and estimate the performance of IVN communication models (Demmeler and Giusto, 2001); H.Moon et al. implemented a heating ventilation and air-conditioning control system based on AUTOSAR architecture and tested it using MATLAB and SIMULINK (Moon et al., 2009). The systems they examined were detailed and contained both software and hardware information. These works used testing to detect the systems. The testing of a system is to input a variety of test cases to test the correctness of the output of the system. It is difficult to enumerate all possible states of the system, so testing is incomplete.

Other studies are directly implemented an IVN system on integrated components and simulate and test electronic signal on hardware. For example, G. Feng et al. implemented and tested a CAN system with electronic nodes (Feng et al., 2010); F. Baronti et al. had designed and implemented a FlexRay protocol component to verify the fault tolerance of IVN systems (Baronti et al., 2011). These studies implemented an IVN system on physical devices, and then connected them to a PC for software-based testing. This method requires both software and hardware support, and the testing is based on hardware electrical signals. The test effectiveness is directly dependent on test cases. Although these studies are closer to a real system, they lack completeness and flexibility.

Some studies only test and analyze the system itself. For instance, S. Anssi et al. focused on analyzing the scheduling capability of an AUTOSAR system (Anssi et al., 2011). Although these studies can be used to analyze and test IVN systems concretely and intuitively, a completed set of test cases is difficult to design for checking the properties of a system. It is also hard to precisely check concurrent behaviors and logic errors in the system design phase.

Compared with these studies, although our model has no operating system or physical protocol, this highly abstract model can reflect the communication behavior of the system in the system design phase and verify timed properties accurately and completely. Our proposed framework has the flexibility to build IVN systems with different protocols and topologies. Other researchers take multi-protocol communication into account, but they focus on improving the performance of gateways with CAN, LIN, FlexRay and MOST (Seung-Han et al., 2008; Kim et al., 2015; Schmidt et al., 2010; Zhao et al., 2010). The common idea is to provide efficient scheduling algorithms to the gateway. For example, Kim et al. examine the implementation of a CAN–FlexRay gateway using message-mapping (Kim et al., 2015). In our work, the gateway is used to forward messages, which only has the most basic function of forwarding messages and a simple scheduling mechanism. Our focus is on the effect of multiple protocols on timed properties. Although gateways can affect the communication efficiency of the system, even without gateways, the message transmission time between multiple protocols is different from a partitioned system, as mentioned the example in Section 2.3. Thus, our work verified the timed property effectively.

6.2. Model checking of IVN systems

Model checking is another effective method for verification of safety-critical systems. This method is exhaustively and automatically applied to check properties by searching all states of the system. The properties can be specified in temporal logic and verified precisely. There are many works on checking communication protocols (Gerke et al., 2010; Pinto et al., 2007; Saha and Roy, 2007). They modeled protocols and verified the properties from their specifications, such as the fault-tolerance on the FlexRay physical layer, the start-up process of FlexRay, and the error handling and fault tolerance of timed-triggered CAN. These work by focusing on analyzing and verifying the protocol itself. Our work verifies the IVN system design with abstracted applications.

For the software, some works (Waszniowski and Hanzálek, 2008; Chen and Aoki, 2011; Huang et al., 2011; Fang et al., 2012) aimed at verifying that implementations of the system are consistent with software standards. J. Chen et al. designed a model based on the OSEK/VDX operating system and verified that the model meets OSEK specification using SPIN, and generated test cases us-

ing the model (Chen and Aoki, 2011). L. Fang et al. proposed a formal model of a real-time, multi-core AUTOSAR operating system and developed a test case generator and a test program generator to complete system testing (Fang et al., 2012). Y. Huang et al. implemented a formal model of OSEK/VDX OS with CSP language and verified the model using PAT (Huang et al., 2011). These studies developed formal operating system models, and these models are used to generate exhaustive test cases to improve system testing. The studies did not consider communication protocols, but they analyzed task behaviors in the application layer. We do not consider task scheduling in the application model, and tasks are simplified as message distributors, which send messages in different cycles.

In addition, some studies focus on interactive behaviors between ECUs in a single protocol (Waszniowski et al., 2009; Pan et al., 2014). L.Waszniowski et al. established a complete CAN system model with the OSEK/VDK operating system and showed a case study to verify the timed properties of the system (Waszniowski et al., 2009). C.Pan et al. showed a CAN protocol model to verify some primary properties using the UPPAAL model checker and considered an application model with a scheduling algorithm to fix some properties that were unsatisfied (Pan et al., 2014). Our previous work only modeled and verified the communication behaviors of a FlexRay system (Guo et al., 2013). However, an IVN system is extremely complicated, containing multiple protocols, several gateways, and many applications. It is difficult to directly model the complete IVN system and efficiently verify properties. Most researches consider a single protocol in the IVN system, and few studies consider a complete IVN system architecture with multiple protocols. Our framework can check IVN systems with multiple protocols. Furthermore, the modularization of our framework means it is reusable for modeling and verifying different IVN system topologies.

7. Conclusion and future work

In this paper, we have presented a framework for modeling and verifying communications in IVN systems that operate the CAN and FlexRay protocols. The framework was built through a two-staged abstraction on the basis of the protocol specifications of CAN and FlexRay while preserving the essential functionalities of communications. The contribution of this work is to describe an appropriate abstraction for modeling IVN systems and realize the associated framework. In the framework, CAN and FlexRay are taken into account and reusable modules enable the verification of a variety of application designs given the appropriate configuration. The framework has been evaluated from four aspects. At first, we have discussed the validity of the abstraction. The framework was proposed based on the two-stage abstraction. In the first stage, we abstracted the architecture of IVN systems, where subnetworks are abstracted as environments. Thus, we restored the subnetworks with different topologies and checked the response time of messages. The results suggest that the abstraction can preserve the timed properties outside of subnetworks in the IVN system. In the second stage, we abstracted the CAN and FlexRay specifications. We listed some properties about transmission schemes that can be preserved, and the signal-level behaviors and error handling that can not be preserved in the framework. Then, the framework was applicable and reusable for different IVN system designs. We implemented IVN systems with three typical topologies and checked timed properties in the three cases. The checking results accorded with the characteristic of the topologies; for different IVN system design, the modules of the framework were unchanged, except for the application layer, *Environment* and *Forwarder* modules. Besides, the performance of the framework has been evaluated. Model checking can exhaustively verify properties and give out precise results, but the capability and efficiency are limited. In future work, we will consider other important properties associated with the communication within IVN systems. For instance, we attempted to check whether all messages were received by the destination node, a property related to the safety of the IVN system. However, the check did not allow us to determine how many messages were lost. Thus, we will apply statistical model checking to overcome this issue.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Altran Technologies, 2005. FlexRay communications system protocols specification V.3.0.1.

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

Armbruster, M., Zimmer, E., Lehmann, M., De, R.R., 2006. Affordable X-By-Wire technology based on an innovative, scalable E/E platform-concept. In: 2006 IEEE 63rd Vehicular Technology Conference, pp. 3016–3020.

Baronti, F., Petri, E., Saponara, S., Fanucci, L., Roncella, R., Saletti, R., 2011. Design and verification of hardware building blocks for high-speed and fault-tolerant in-vehicle networks. IEEE Trans. Ind. Electron. 58 (3), 792–801.

Behrmann, G., David, A., Larsen, K., 2004. A Tutorial on UPPAAL. In: Formal Methods for the Design of Real-Time Systems, 3185. Springer, Berlin, Heidelberg, pp. 200–236.

Bel Mokadem, H., Bérard, B., Gourcuff, V., De Smet, O., Roussel, J.M., 2005. Verification of a timed multitask system with UPPAAL. In: Emerging Technologies and Factory Automation (ETFA'05), pp. 921–932.

Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W., 1995. Uppaal – a tool suite for automatic verification of real-time systems. In: Workshop on Verification and Control of Hybrid Systems III, pp. 232–243.

Bosch, 1991. CAN specification version 2.0.

Chen, J., Aoki, T., 2011. Conformance testing for OSEK/VDX operating system using model checking. In: Proceedings – Asia-Pacific Software Engineering Conference, APSEC, pp. 274–281.

Coles, Z.A., Beyerl, T.A., Augusma, I., Soloiu, V., 2016. From sensor to street â intelligent vehicle control systems. Pap. Publ. 5, 52–58.

Demmeler, T., Giusto, P., 2001. A universal communication model for an automotive system integration platform. In: Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001, pp. 47–54.

Dominguez-Garcia, A.D., Kassakian, J.G., Schindall, J.E., 2006. Reliability evaluation of the power supply of an electrical power net for safety-relevant applications. Reliab. Eng. Syst. Saf. 91 (5), 505–514.

Elmenreich, W., Krywult, S., 2005. A comparison of fieldbus protocols: LIN 1.3, LIN 2.0, and TTP/A. In: 2005 IEEE Conference on Emerging Technologies and Factory Automation, pp. 747–753.

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

Feng, G.S., Zhang, W., Jia, S.M., Wu, H.S., 2010. CAN bus application in automotive network control. In: 2010 International Conference on Measuring Technology and Mechatronics Automation, ICMTMA 2010, 1, pp. 779–782.

Gerke, M., Ehlers, R., Finkbeiner, B., Peter, H., 2010. Model checking the FlexRay physical layer protocol. Formal Methods Ind. 132–147.

Grimm, K., 2003. Software technology in an automotive company – major challenges. In: 25th International Conference on Software Engineering, 2003. Proceedings., 6, pp. 498–503.

Guo, X., Aoki, T., Chiba, Y., Lin, H., 2017. A reusable framework for modeling and verifying in-vehicle networking systems in the presence of CAN and FlexRay. In: 24th Asia-Pacific Software Engineering Conference (APSEC), pp. 140–149.

Guo, X., Lin, H.-H., Yatake, K., Aoki, T., 2013. An UPPAAL framework for model checking automative systems with FlexRay protocol. In: Formal Techniques for Safety-Critical Systems (FTSCS'13), 419, pp. 36–53.

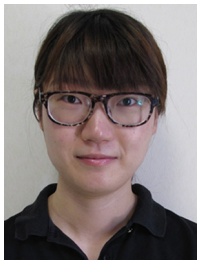
Hiraoka, T., Eto, S., Nishihara, O., Kumamoto, H., 2004. Fault tolerant design for X-By-Wire vehicle. In: SICE 2004 Annual Conference, pp. 1940–1945.

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

Johansson, K.H., Martin, T., 2005. Vehicle applications of controller area network. In: Handbook of Networked and Embedded Control Systems. Birkhäuser Boston, pp. 741–765.

Kim, M.-H., Lee, S., Lee, K.-C., 2015. Performance evaluation of node-mapping-based Flexray-CAN Gateway for in-vehicle networking system. Intell. Autom. Soft Comput. 21 (2), 251–263.

Leen, G., Heffernan, D., 2002. Expanding automotive electronic systems. Computer 35 (1), 88–93.
Moon, H., Kim, G., Kim, Y., Shin, S., Kim, K., Im, S., 2009. Automation test method for automotive embedded software based on autosar. In: 4th International Conference on Software Engineering Advances, ICSEA 2009, Includes SEDES 2009: Simposio para Estudantes de Doutoramento em Engenharia de Software, pp. 158–162.
Pan, C., Guo, J., Zhu, L., Shi, J., Zhu, H., Zhou, X., 2014. Modeling and verification of CAN bus with application layer using UPPAAL. Electron. Notes Theor. Comput. Sci. 309, 31–49.
Park, I., Sunwoo, M., 2011. Flexray network parameter optimization method for automotive applications. IEEE Trans. Ind. Electron. 58 (4), 1449–1459.
Pinto, A., Carloni, L.P., Sangiovanni-Vincentelli, A.L., 2007. Verification of FlexRay start-up mechanism by timed automata. In: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software – EMSOFT '07, p. 21.
Saha, I.S.I., Roy, S.R.S., 2007. A finite state analysis of time-triggered CAN (TTCAN) protocol using spin. In: 2007 International Conference on Computing: Theory and Applications (ICCTA'07).
Sangiovanni-Vincentelli, A., 2003. Electronic-system design in the automobile industry. IEEE Micro 23 (3), 8–18.
Schmidt, E.G., Alkan, M., Schmidt, K., Karakaya, U., 2010. Performance evaluation of FlexRay/CAN networks interconnected by a gateway. In: International Symposium on Industrial Embedded Systems (SIES), pp. 209–212.
Seo, H.S., KIM, B., 2013. Design and implementation of a UPNP-CAN gateway for automotive environments. Int. J. Automot. Technol. 14 (1), 91–99.
Seung-Han, K., Suk-Hyun, S., Jin-Ho, K., Tae-Yoon, M., Chang-Wan, S., Sung-Ho, H., Jae Wook, J., 2008. A gateway system for an automotive system: LIN, CAN, and FlexRay. In: Industrial Informatics (INDIN'08), pp. 967–972.
Waszniowski, L., Hanzálek, Z., 2008. Formal verification of multitasking applications based on timed automata model. Real-Time Syst. 38 (1), 39–65.
Waszniowski, L., Krákora, J., Hanzálek, Z., 2009. Case study on distributed and fault tolerant system modeling based on timed automata. J. Syst. Softw. 82 (10), 1678–1694.
Wolf, M., Weimerskirch, A., Christof, P., 2004. Security in automotive bus systems. In: Proceedings of the Workshop on Embedded Security in Cars (ESCAR)'04, pp. 1–13.
Zhao, R., Qin, G.H., Liu, J.Q., 2010. Gateway system for CAN and FlexRay in automotive ECU networks. In: International Conference on Information, Networking and Automation (ICINA 2010), 2, pp. 49–53.
Zhu, X.F., Wang, J.D., Li, B., Zhu, J.W., Wu, J., 2009. Methods to tackle state explosion problem in model checking. In: 3rd International Symposium on Intelligent Information Technology Application, IITA 2009. IEEE, pp. 329–331.



Xiaoyun Guo is presently a researcher in the School of Information Science at the Japan Advanced Institute of Science and Technology (JAIST). She received here Ph.D from JAIST (2019). Her research interests include model checking, automotive systems, and communication protocols.

Hsin-Hung Lin is received his Ph.D. degree in information science from JAIST in 2011. He was a post-doctoral research fellow at the Institute of Information Science, Academia Sinica in 2012, and a research fellow in the Graduate School of Information Science and Electrical Engineering, Kyushu University, form 2013 to 2015. He is currently a post-doctoral research fellow in the Institute of Information Science, Academia Sinica. His research interests include formal methods, formal verification, model checking, and their applications to software engineering.

Toshiaki Aoki is a professor in the School of Information Science, JAIST(Japan Advanced Institute of Science and Technology). He received his B.S. from the Science University of Tokyo (1994), and his M.S. and Ph.D. from JAIST (1996, 1999). His research interests include model checking, theorem proving, formal specification, embedded systems, automotive systems and the industrialization of formal methods.