An UPPAAL Framework for Model Checking Automotive Systems with FlexRay Protocol

Xiaoyun Guo[1], Hsin-Hung Lin[2(✉)], Kenro Yatake[1], and Toshiaki Aoki[1]

[1] School of Information Science,
Japan Advanced Institute of Science and Technology, Ishikawa, Japan
{xiaoyunguo,k-yatake,toshiaki}@jaist.ac.jp
[2] School of Information Science and Electrical Engineering, Kyushu University, Fukuoka, Japan
h-lin@ait.kyushu-u.ac.jp

Abstract. This paper introduces a method and a framework for verifying automotive system designs using model checking. The framework is based on UPPAAL, a timed model checker, and focuses on checking automotive system designs with FlexRay communication protocol, a de facto standard of automotive communication protocols. The framework is composed of FlexRay model and application model where the former is built by abstractions to the specifications of FlexRay protocol. In the framework, FlexRay model is reusable for different application models with appropriate parameter settings. To the best of our knowledge, the framework is the first attempt on model checking automotive system designs considering communication protocols. Checking of core properties including timing properties are conducted to evaluate the framework.

1 Introduction

Automotive systems mainly adopt electronic control units (ECUs) to realize X-by-wire technology [10]. With the X-by-wire technology, requirements or functionalities which were not mechanically realizable are possible. Generally, ECUs in an automotive system follow communication protocols to communicate with each other through one or multiple buses. Since communication protocols greatly affect the performance of an automotive system, protocols which can support high transmission rate while still having reliability are demanded. Recently, FlexRay communication protocol is considered the de facto standard of automotive communication protocols [1,13]. FlexRay supports high transmission rate up to 10 Mbs while still having fault-tolerance abilities. These characteristics make FlexRay especially suitable for safety critical systems.

Increasing requirements for safety, driving assistance, etc., result in more complexity in the development of automotive systems. More ECUs are required in automotive systems and hence the need for handling heavy communications. Therefore, validation and verification of automotive systems became much harder. In industry, integration platform based solutions are proposed to support

C. Artho and P.C. Ölveczky (Eds.): FTSCS 2013, CCIS 419, pp. 36–53, 2014.
DOI: 10.1007/978-3-319-05416-2_4, © Springer International Publishing Switzerland 2014

automotive system design processes [7,14,15]. Integration platforms provide virtual simulation and testing for automotive system designs and implementations and thus save the cost of testing on devices. Although integration platforms can perform early phase analysis, behavioral analysis as well as verification of design models is hard to conduct because simulation and testing only focus on specific signals or nodes in a system. On the other hand, timed model checking techniques are proven effective on verification of real time systems [2,9]. Therefore, introducing timed model checking on verifying automotive system designs is considered appropriate and necessary.

This paper proposes a method for verifying design models of automotive systems using timed model checking technique. The method considers automotive systems with FlexRay protocol and focuses on communications between ECUs. Based on the method, a framework is implemented on UPPAAL, a timed model checker [3]. UPPAAL has a nice graphical user interface for precisely describing time constrained behaviors and is widely used in verifying time critical systems. However, when considering automotive systems with FlexRay protocol, it is recognized that the behaviors of FlexRay and tasks affect each other all the time. This phenomena is difficult to be precisely modeled using primitive channel synchronizations provided by UPPAAL. Therefore, we model an automotive system as the combination of FlexRay model and application model, where the former is reusable with different parameter settings of systems. Developers can build an automotive system design on application model and verify it with FlexRay model plus proper parameter settings.

The proposed model will focus on verification of design models, especially behavior and timing related to message transmissions. Three steps of abstractions are applied on the FlexRay communication protocol to remove parts and behaviors not in focus, and then build FlexRay model. To evaluate the framework, experiments on simple systems are demonstrated to examine if the framework precisely models the behavior of normal transmissions in FlexRay protocol and its ability of timing analysis for automotive system designs.

2 Related Work

Practically, automotive systems are tested and validated using integration platform solutions in the industry [7,14,15]. Integration platforms provide virtual environments for simulation and analysis of automotive systems. However, testing or simulation can only focus on specific signals or nodes in a system so that high level analyses such as behavioral analysis are difficult. Compared to integrated platforms, our framework focuses on behavior analysis and verification with time, which makes up the above deficiency. Also, to the best of our knowledge, the framework is the first attempt for verification support of automotive system designs considering communication protocol.

Another important issue of automotive systems is scheduling or performance analysis which analyzes expected execution time of applications and sees whether deadlines can be met or not. For FlexRay protocol, dealing with dynamic

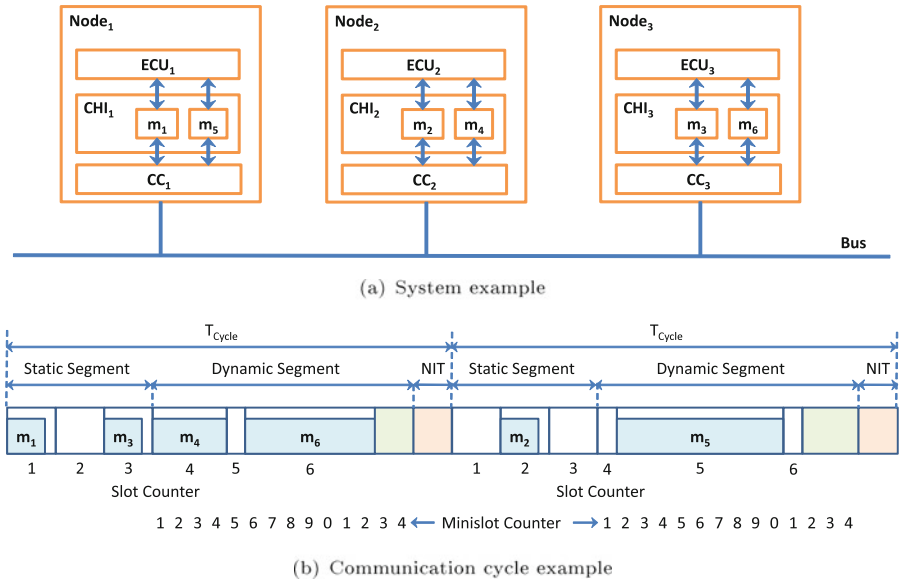


Fig. 1. An example automotive system with FlexRay

segments in FlexRay protocol is the most important as well as difficult issue in approaches of scheduling analysis [8,11,16,17]. Though our work does not explicitly consider scheduling analysis due to simplification on ECUs, the framework is similar to model-based scheduling analysis which has been proved useful on other platforms [4,5]. Therefore, we argue that scheduling analysis is also possible using our framework with some improvements.

For FlexRay protocol itself, correctness of FlexRay protocol is verified in a few aspects. M. Gerke et al. verified the physical layer of FlexRay and proved the fault-tolerance guarantees in FlexRay [6]. J. Malinský and J. Novák verified the start-up mechanism of FlexRay [12]. Based on the results of the above work, our framework assumes the correctness of the physical layer, i.e. encoding and decoding of frames, and the start-up of FlexRay. As a result, we did not implement physical layer and start up mechanism in FlexRay model and focus only on the behavior of abstracted frame transmissions.

3 Automotive Systems with FlexRay Protocol

In the specification of FlexRay, the controller-host interface (CHI) is implementation dependent. For verification purpose, since different implementations need different models, it is necessary to declare which implementation is considered in this paper. In this section, an example of automotive system with FlexRay shown in Fig. 1 is introduced for demonstration.

As Fig. 1(a) shows, an automotive system consists of numbers of nodes connected to a bus for communications with each other. Each node consists of three parts: an ECU, a controller-host interface (CHI), and a communication controller (CC). In each node, tasks of applications are running on the ECU and send/receive data to/from buffers of the CHI. A CHI contains several buffers designated for specific data streams called frames in FlexRay. For every frame, a sending and a receiving buffer are specified. The sending buffer of a frame is implemented in the CHI where the ECU of same node has tasks designated to send data by the frame. The receiving buffer of a frame is implemented in the CHI where the ECU of same node has tasks designated to receive data by the frame. When an automotive system is executing, the CC of a node counts the time in each cycle and sends a frame from the corresponding buffer to the bus at the designated time. The CC also receives a frame and writes to the corresponding buffer if the frame is designated to be received by the node. Note that only one frame is allowed to be sent and received at the same time. It should also be noted that in a node, the status of the CC, i.e. current number of cycles, current number of slots, etc., is accessible to tasks in the ECU through the CHI and thus makes more complicated behaviors possible. In Fig. 1(a), The system has three nodes, $Node_1$, $Node_2$, and $Node_3$. Six frames are defined and sending buffers are specified in the corresponding CHIs: m_1 and m_5 in CHI_1, m_2 and m_4 in CHI_2, m_3 and m_6 in CHI_3[1].

Figure 1(b) demonstrates a two cycle instance of communications of the system shown in Fig. 1(a). Communications in FlexRay are performed based on periodic cycles. A cycle contains two time intervals with different policies for accessing the bus: static segment and dynamic segment[2]. The lengths of the two segments are fixed in every cycle and the time in a cycle is counted using slots: static slots and dynamic slots. A static slot has fixed length defined by global configuration parameter *gdStaticSlot* as a common time unit called macrotick (MT) for all nodes in a system and the length of a static segment of is defined by global configuration parameter *gNumberOfStaticSlots*. On the other hand, dynamic slots are flexible and composed of several minislots. A minislot is the basic unit of a dynamic segment and its length is defined by global configuration parameter *gdMinislot* as macroticks. The length of a dynamic segment is then defined by global configuration parameter *gNumberOfMinislots*. The index of a frame should be defined to map with a slot number therefore a frame will be sent at designated time interval, i.e. slot, in a cycle.

In Fig. 1(b), the index of a frame is set to the same slot number for convenience. Frame m_1, m_2, and m_3 are set to static slots, slot 1, 2, and 3. Frame m_4, m_5, and m_6 are set to dynamic slots, slot 4, 5, and 6. In the static segment of the first cycle, frame m_1 and m_3 are sent in slot 1 and slot 3. Though frame m_2 is not sent, the time interval of slot 2 is still elapsed with no transmission.

[1] Receiving buffers are not shown in the figure.

[2] Here we ignore symbol window (SW) and network idle time (NIT). The former is optional and the latter is for adjustment of cycle length. Both SW and NIT do not affect communications in automotive system designs.

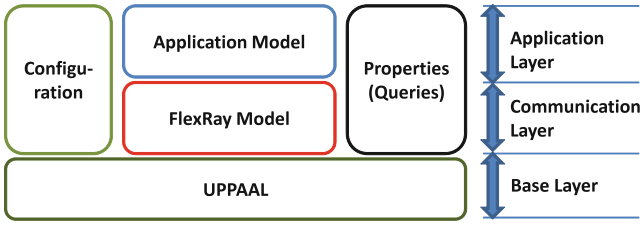


Fig. 2. The framework

In the dynamic segment of the first cycle, m_4 and m_6 are sent in slot 4 and slot 6. Frame m_4 has the length of four minislots and m_6 has the length of seven minislots. Slot 5 is not sent in first cycle but still occupies the time interval of one minislot. When the maximum slot number is reached but the maximum minislot number is not, the time proceeds with no transmission till the end of the dynamic segment. The second cycle is similar where only m_2 and m_5 are sent.

4 The Framework

Figure 2 shows the structure of the UPPAAL framework for verification of automotive systems with FlexRay demonstrated in Sect. 3. The framework consists of several parts: *UPPAAL engine*, *FlexRay model*, *Application model*, *Configuration*, and *Properties*. The parts of the framework are associated with three layers: base, communication, and application layers. The foundation of the framework is the UPPAAL model checker. *FlexRay model* which models the FlexRay protocol is the main component of the communication layer. *Application model* which represents the design model of an automotive system belongs to the application layer. *Configuration* and *Properties* are associated to both communication and application layers: *Configuration* contains parameters relating to both FlexRay and application models; *Properties* specify queries for verification of both FlexRay and application models in UPPAAL. FlexRay model and application model, which are the main components of the framework, will be described in Sects. 4.1 and 4.2 separately. *Configuration* and *Properties* will be mentioned within examples in Sects. 4.3 and 5.

4.1 FlexRay Model

The specifications of the FlexRay communication protocol include details of implementations on hardwares irrelevant to verification of design models of automotive systems. Therefore, to build FlexRay model in our framework, abstractions are needed to trim off irrelevant parts and behaviors. Generally, the abstractions are processes of modeling the specifications of the FlexRay protocol based on our understanding of the FlexRay protocol and our knowledge/experiences of using UPPAAL. We divide the processes of the abstractions

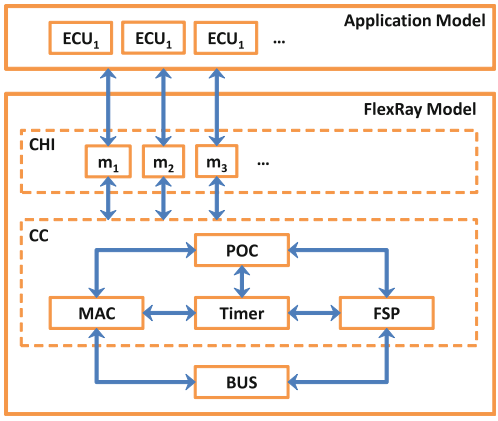


Fig. 3. The structure of FlexRay model

into three steps: (1) essential component selection, (2) functionality reduction, and (3) state space reduction. Figure 3 shows the structure of the FlexRay model after the abstraction. The details of the three steps abstraction are described as follows.

Essential Component Selection. For the purpose of verifying the design model of an automotive system, we only focus on functionalities relating to sending and receiving frames when building FlexRay model. The first step is then to select essential and necessary components providing frame transmission functionalities. Since we only focus on design level verification, specifications regarding low level behaviors such as clock synchronization and frame encoding/decoding are out of focus. We also assume that there is no noise interference in transmissions[3]. Therefore, we only need three components in FlexRay protocol: *protocol operation control* (*POC*), *media access control* (*MAC*), and *frame and symbol processing* (*FSP*). *POC* monitors overall status of CC and manages other components. *MAC* is responsible for sending frames in corresponding sending buffers of CHI at specific times in each cycle. *FSP* is responsible for receiving frames and storing data to receiving buffers in CHI for tasks in ECUs to read. Besides *POC*, *MAC*, and *FSP*, we also need *Timer* which helps monitoring of timing in each cycle and slot. *Timer* is not treated as a component in the specifications of the FlexRay protocol but we have to build it since timer is used everywhere in almost all components of a CC. The bus is implemented as a variable accessible to *MAC* and *FSP*. The sending/receiving of frames is then represented by writing/reading data to/from the bus variable.

[3] Generally, FlexRay only captures and throws errors. An application has the responsibility to handle errors thrown by FlexRay. Though not in the scope of this paper, if transmission errors are of interest, they can be modeled by adding error situations/states explicitly in FlexRay model.

Functionality Reduction. After the first step of the abstractions, the selected components *POC*, *MAC*, and *FSP* still have irrelevant behaviors which do not participate in activities related to frame transmissions. Also, the irrelevant behaviors may still cooperate with components already removed in the first step of the abstractions. Therefore, the second step is to remove the irrelevant behaviors and functionalities of the selected components. In the framework, we only focus on regular transmissions of frames and therefore only the normal state of *POC* is meaningful and other states are ignored. Furthermore, we consider that the clock is synchronized between nodes and there is no external interference in normal transmissions. This results that some functionalities of CC become unnecessary for FlexRay model. For example, functionalities such as adjustments for reactions on transmission errors, e.g. fault tolerance feature of the bus, can be ignored. Also, similar functionalities mainly related to error managements in other components and CHI are ignored. Note that our most priority of modeling the FlexRay protocol is to fulfill the need of timing analysis, which is not required to consider situations with errors in the first place. Therefore, the modeling process is more like picking up the traces of successful frame transmissions but trimming off error processing behaviors.

State Space Reduction. After the above two steps of the abstractions, FlexRay model looks simple considering the number of explicit states and transitions. However, the complexity is still high and hardly acceptable since there are many variables especially clocks in FlexRay model. In some cases even the size of an application is not considered large, UPPAAL suffers from state explosion when checking properties. Therefore, further abstraction is necessary to reduce the state space while the behaviors of frame transmissions in the FlexRay protocol is still precisely modeled. By reviewing the above two steps of abstractions, recall that there are two assumptions of FlexRay model in the framework: (1) all nodes are synchronized all the time; (2) there is no error during frame transmissions. With (1), all nodes start a cycle at the same time; with (2) all nodes finish a cycle at the same time. That is, no node is going to be late because of transmission errors. Therefore, it is reasonable to conclude that we do not need a CC for every node, i.e. one CC is enough. This helps us to remove the complicated behaviors which only synchronize the clocks of nodes of a system. Furthermore, we also cancel the process of counting minislots in dynamic segment and instead calculating the number of minislots directly using lengths of dynamic frames and related parameters. This helps avoiding small time zones not meaningful in checking properties. Most properties concern the timing of the start and the end of a frame transmission but not in the middle of a frame transmission.

FlexRay model constructed after three steps of abstraction is shown in Figs. 4 and 5, where *MAC* is separated into *MAC_static* and *MAC_dynamic*. An example will be given in Sect. 4.3 for demonstrating how FlexRay model works in cooperation with *Application model* explained in Sect. 4.2.

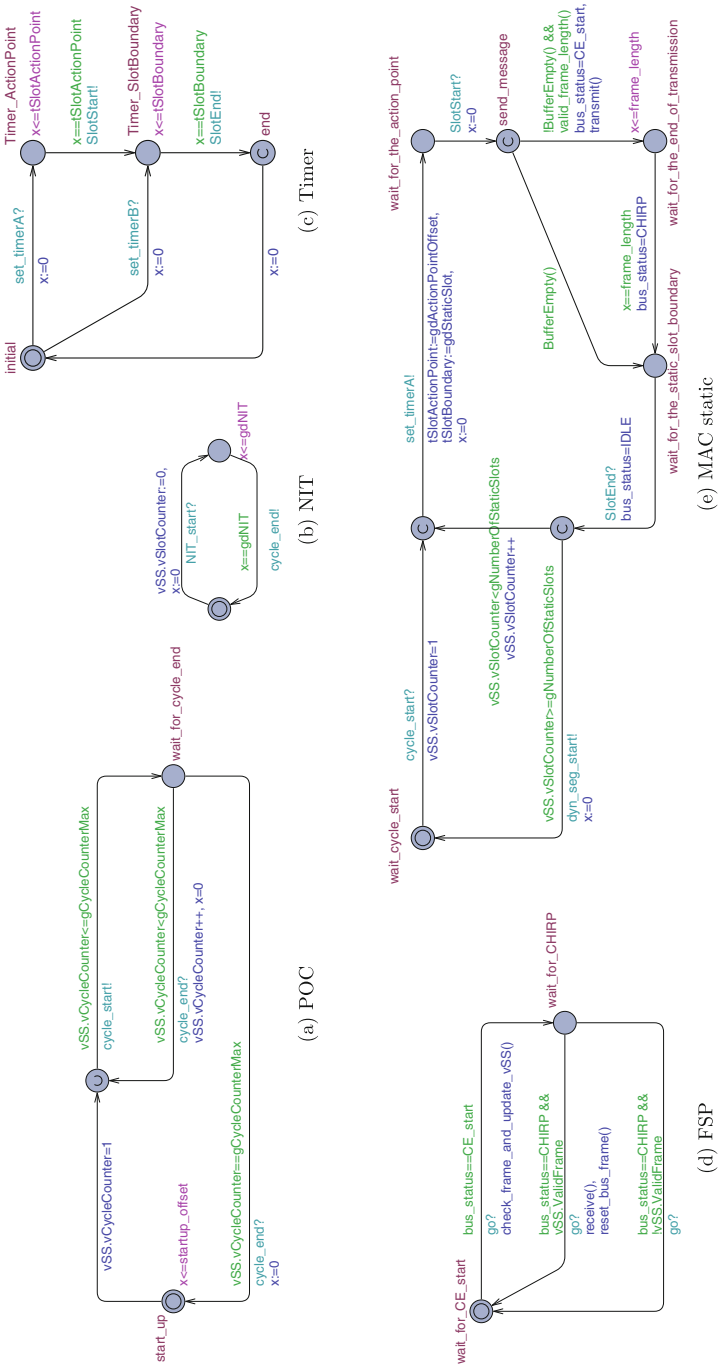


Fig. 4. FlexRay model (POC, NIT, Timer, FSP, MAC static)

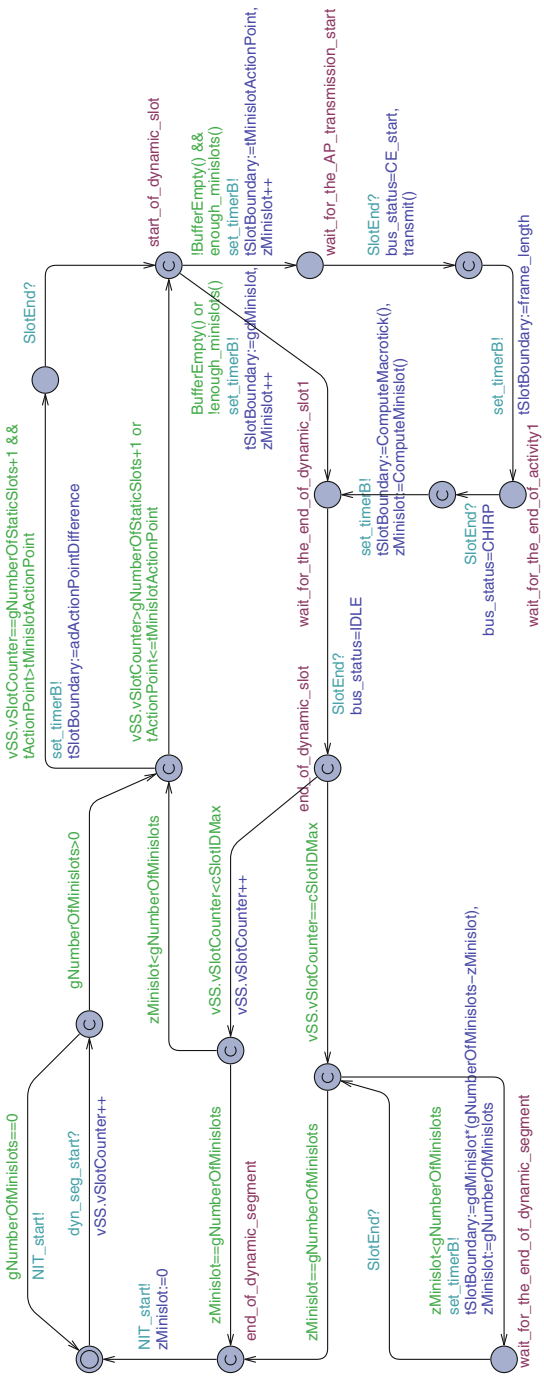


Fig. 5. FlexRay model (MAC dynamic)

4.2 Application Model

Application model represents ECUs in an application and thus consists of multiple tasks. As shown in the upper part of Fig. 3, application model accesses the buffers in CHIs to communicate with CCs for sending/receiving frames. Since *Application model* tightly depends on actual automotive systems to be designed by developers, we leave most of the jobs to developers in building *Application model* and give only simple directions on how to use *FlexRay model* of the framework.

Only One Task in an ECU. For simplicity, in this paper we build one module in UPPAAL to represent one task and an ECU only has one task. Therefore we can omit modeling of schedulers in ECUs. Developers have to build a scheduler module when scheduling in an ECU is considered necessary.

Use of Functions to Access Buffers in CHIs. In an automotive system with FlexRay protocol, tasks in different nodes cannot communicate directly but through FlexRay protocol, i.e. *FlexRay model* of the framework. Therefore, when sending data, a task has to write data to the corresponding sending buffer in the CHI of the same node, and let the CC do the transmissions. When receiving data, the process is similar but in the reverse order. To make things simple, we prepare functions for reading and writing data from and to specified buffer. Developers only need to put these functions as actions on transitions of tasks and insert proper parameters. The functions are defined as follows:

```
void write_msg_to_CHI(t_msg_slot msg, int value, int len);
int read_msg_from_CHI(t_msg_slot msg);
void clean_send_buffer_CHI(t_msg_slot msg);
void clean_receive_buffer_CHI(t_msg_slot msg);
```

Since we do not focus on the contents of the data in a frame, the data is represented simply by integer type and may be ignored. Note that `msg` of type integer is the index of a frame as well as the index of the corresponding buffer; `len` is the length of the frame in macroticks. Note that reading the data from a buffer does not clean up the buffer so there are also functions for cleaning buffers. Developers have to clean a buffer by themselves using buffer cleaning functions.

4.3 Example

In this section, a simple sender/receiver example [18] will be demonstrated to show how the design model of an automotive system built as *Application model* looks like and how frames are transmitted by *FlexRay model*. This example consists of tasks having simple behaviors so that we can focus on reading/writing buffers, frame transmissions, and parameter settings of the system. Figure 6 shows the plan of assigning indexes of frames. There are ten messages/frames

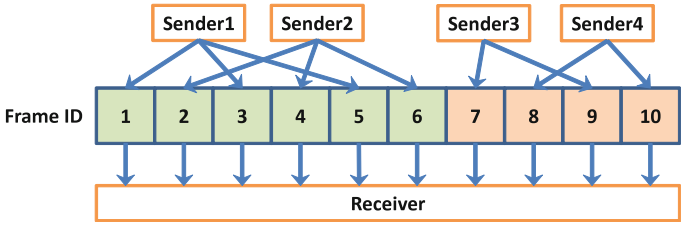


Fig. 6. Frame setting of the sender/receiver example (SR1)

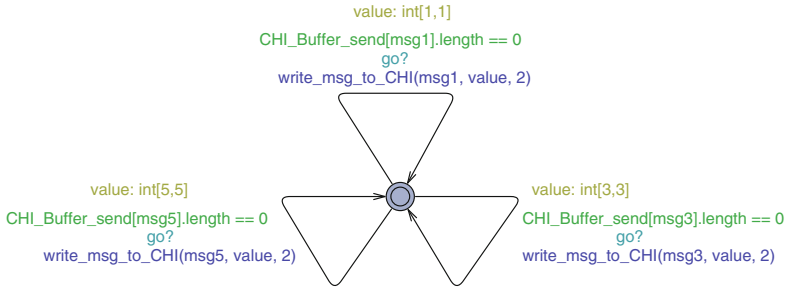
indexed from 1 to 10. The frames are used in five ECUs/tasks, `Sender1`, `Sender2`, `Sender3`, `Sender4`, and `Receiver`. Which task sends/receives which frame can be easily recognized by directions of the arrows. For example, `Sender1` is designed to send frames 1, 3, and 5, and `Receiver` is designated to receive all frames. Frames 1 to 6 are static frames and frames 7 to 10 are dynamic frames.

```
typedef int[1,cSlotIDMax] t_msg_slot;
const t_msg_slot msg1= 1;
...
const t_msg_slot msg10= 10;
```

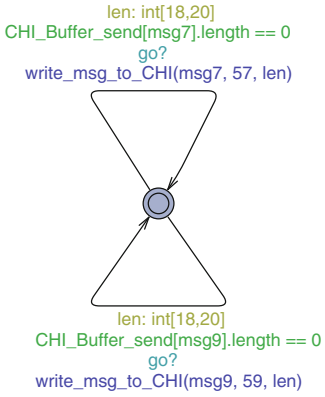
As mentioned in Sect. 4.2, the indexes of frames are the same as the indexes of buffers. Also, the indexes of frames indicate the slot numbers of the communication cycles in *FlexRay model*. Below shows the major parameters of the example. The unit of parameters is macrotick except the first four parameters.

```
int gCycleCounterMax=6; //max. number of cycle
int gNumberOfStaticSlots=6; //number of static segment slots
int gNumberOfMinislots=32; //number of dynamic segment minislots
int cSlotIDMax=10; //max. number of slot ID
int gdNIT=4; //period of NIT (in macrotick)
int gdActionPointOffset=2; //static offset
int gdStaticSlot=5; //number of macroticks in a static slot
int gdMinislotActionPointOffset=1; //offset of minislot
int gdMinislot=3; //number of macroticks in a minislot
```

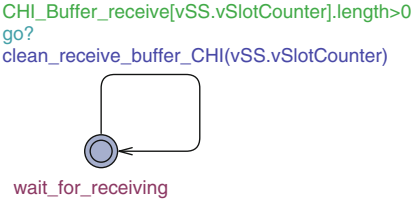
Figure 7 shows modules of `Sender1`, `Sender3`, and `Receiver` in *Application model*, where `Sender2` and `Sender4` are similar and skipped. Note that `go?` is the reception of the urgent channel `go`. Urgent channel always sends a signal immediately without delay when a transition with `go?` is fired. For example, by using `go?`, `Sender1` watches the status of related buffers and writes data to a buffer immediately when the buffer is detected empty. On the other hand, `Sender3` sends dynamic frames whose length vary from 18 to 20 macroticks. In the framework we define global variables to represent buffers (i.e. status of CHIs) and slot status (i.e. status of CCs). *Application model* can access the status of CHIs and CCs through global variables `CHI_Buffer_send`, `CHI_Buffer_receive`, and `vSS`.



(a) Sender1



(b) Sender3



(c) Reciever

Fig. 7. Selected tasks of the sender/receiver example (*SR1*)

```
typedef struct {
   int[0,MaxDataValue] data;
   int[0,pPayloadLengthDynMax] length;
} Buffer;
Buffer CHI_Buffer_send[cSlotIDMax+1]; //sending buffers
Buffer CHI_Buffer_receive[cSlotIDMax+1]; //receiving buffers

typedef struct{
    int[0,gCycleCounterMax] vCycleCounter; //current cycle
    int[0,cSlotIDMax] vSlotCounter; //current slot
} T_SlotStatus;
T_SlotStatus vSS; //slot status of CC
```

In this system, sending buffers can be considered always filled for convenience. Therefore we may focus on the flow of a static or dynamic frame transmission in FlexRay model shown in Figs. 4 and 5. When the system starts, FlexRay model starts from *POC*. Like the ordering of segments in a communication cycle shown in Fig. 1(b), *POC* counts the number of cycles and activates *MAC_static* for transmitting static frames in the static segment. When the static segment

ends, *MAC_static* activates *MAC_dynamic* to start transmitting dynamic frames in the dynamic segment. When the dynamic segment ends, *NIT* is activated and proceeds to the end of the current cycle, then *POC* takes control again to start another cycle.

In the static segment, *MAC_static* calls *Timer* to set the times of the start and the end of a static slot. In dynamic segment, *MAC_dynamic* has to see if there is a dynamic frame to be sent to decide the number of minislots to proceed. If there is a dynamic frame to be sent, the frame is sent by writing the data of the frame, which is the content of the corresponding sending buffer, to the bus variable. The slot counter `vSS.vSlotCounter` is increased by 1 and the minislot counter `zMinislot` is computed according to the length of the frame. If there is no dynamic frame to be sent, both `zMinislot` and `vSS.vSlotCounter` are just increased by 1. When the maximum slot number is reached, the *MAC_dynamic* will just proceed the remaining minislots to the maximum number of minislots, and then end the dynamic segment. Note that in both *MAC_static* and *MAC_dynamic*, the bus status is set to *EC_start* at the start of a frame transmission and the bus is set to *CHIRP* when the transmission ends. As the receiving side of frame transmissions, *FSP* monitors bus status all the time in a communication cycle and starts to receive a frame when *CE_start* is detected. The end of a frame transmission is at the point that *CHIRP* is detected by *FSP* and the data of the received frame is written to the corresponding receiving buffer.

5 Evaluation of the Framework

In this section, the framework is evaluated by checking some properties on two example applications [18]. Firstly, the sender/receiver example (*SR1*) demonstrated in Sect. 4.3 is used to verify core properties related to frame transmissions of FlexRay protocol to see whether the framework is built right on the scope of frame transmissions. Then we introduce another sender/receiver example (*SR2*) to illustrate possible usage of the framework for timing analysis. Both examples are checked by using UPPAAL 4.1.14 on a machine of following specifications: Windows 8 with Intel i5 2.3GHz and 8GM RAM. Memory usage and CPU times in checking *SR1* are listed in Table 1[4].

Table 1. CPU time/state space/memory usage in checking *SR1*

Query	CPU time (s)	States explored	Memory usage (MB)
q1	1.8	343,821	27.5
q2	14.2	1,121,950	107.3
q3	14.5	1,118,872	112.1
q4	4.8	470,860	106.0
q5	7.2	470,860	106.7

[4] We used `verifyta` in command-line with `-u` option.

Is the Framework Built Right? For *SR1*, we give and check some properties/queries based on the specifications of the FlexRay protocol relating to frame transmissions. The results give the hints for evaluating whether *FlexRay model* of the framework is built right, i.e. follows the specifications of the FlexRay protocol in the scope of normal frame transmissions. Recall that the designs of the tasks in SR1 make it reasonable for us to keep the focus on only frame transmissions in FlexRay model. The checked queries are listed as follows:

```
q1. A<> forall (i:int[1,10]) (CHI_Buffer_send[i].length>0);
q2. (CHI_Buffer_send[1].length>0) --> (CHI_Buffer_send[1].length==0);
q3. (CHI_Buffer_send[1].length>0) --> (CHI_Buffer_receive[1].length>0);
q4. A[] forall (i:int[1,10]) ((CHI_Buffer_receive[i].length>0)
      imply (vSS.vSlotCounter==i ));
q5. A[] forall (i:int[1,10]) forall (j:int[1,10])
      (CHI_Buffer_receive[i].length>0 && CHI_Buffer_receive[j].length>0)
      imply (j==i);
```

Queries `q1`, `q2`, `q3` check basic functionalities considering the buffers in the CHI. `q1` says all buffers in the system can be filled with data, which means the tasks can successfully write messages to sending buffers. `q2` says the data in a sending buffer will be erased/sent eventually. `q3` says for a sending buffer with data, the corresponding receiving buffer will be filled, which means frames can be correctly delivered by the CC. Since `q1`, `q2`, and `q3` are all satisfied[5], we can confirm that the tasks do communicate through FlexRay model. That is, frame transmissions are performed by *FlexRay model* as expected in the task designs in *Application model*. Then we check queries `q4` and `q5` considering the time of frame transmissions (slots). `q4` says if a receiving buffer has data, the communication cycle is in the interval of the corresponding slot, which means frame transmissions are occurring in the right slot (time interval)[6]. `q5` says there is only one frame being sent in any slot. From the result that `q4` and `q5` are both satisfied, we can confirm that *FlexRay model* does follow the specifications of the FlexRay protocol regarding normal frame transmissions. Therefore, we conclude that we built the framework right under the scope of normal frame transmissions of the FlexRay protocol.

How to Check Timing Properties? One of the major characteristics of the framework is the ability to describe behaviors with time constraints. Here we introduce another sender/receiver example (*SR2*) shown in Fig. 8 [18]. In this system, *Sender* sends a message periodically while *Receiver* receives a message immediately when the receiving buffer is detected having data. Note that *Sender* checks periodically if the sending buffer is filled and only writes data to the buffer when the buffer is empty. The major parameter settings are as follows:

```
int gCycleCounterMax=6; //max. number of cycle
int gNumberOfStaticSlots=3; //number of static segment slots
int gNumberOfMinislots=30; //number of dynamic segment minislots
```

[5] For `q2` and `q3`, all ten messages of indexes 1 to 10 are checked.
[6] Note that *Receiver* receives the data as soon as a receiving buffer is filled.

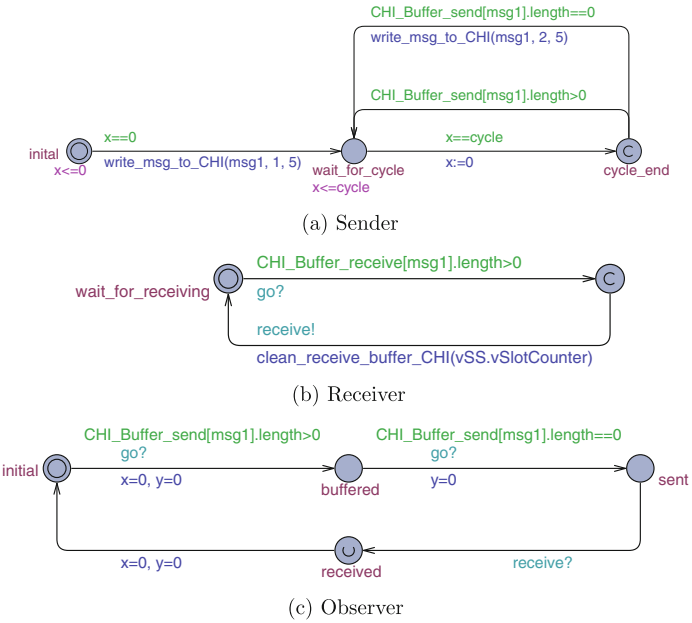


Fig. 8. Another sender/receiver example (SR2)

```
int cSlotIDMax=6;  //maximum number of slot ID
int gdNIT=2; //period of NIT (in macrotick)
int gdActionPointOffset=2; //static offset
int gdStaticSlot=10; // number of macroticks in a static slot
int gdMinislotActionPointOffset=2; //offset of minislot
int gdMinislot=5; // number of macroticks in a minislot
```

Though there is only one frame to be sent/received, the system is defined to have six slots including three static slots. Also, the only frame is set to slot 1, i.e. the first static slot, and the length of the cycle of *Sender* is set to 100 macroticks.

To write property of response time of `msg1`, we built *Observer* to monitor changes in the sending buffer of `msg1`. *Observer* moves from the initial state to state `buffered` once the sending buffer is written by *Sender*. Once the sending buffer is cleaned by FlexRay model when the transmission starts, *Observer* immediately moves to state `sent` and waits *Receiver* to send signal `receive`. The signal `receive` indicates that the transmission is finished and the receiving buffer is written with the data of the received frame. Note that *Observer* has two clocks `x` and `y` where `x` starts counting at the time the sending buffer is written, and `y` starts counting at the time the sending buffer is cleaned. Therefore, by examining the value of clock `x` at state `received` of *Observer*, we can know the response time (macrotick, MT) of `msg1`; by examining the value of clock `y` at state `received` of *Observer*, we can know the frame transmission time of `msg1`. Now we can write some queries about the response times of `msg1`.

```
q1[Y].  A[] (observer.received imply observer.y == 5)
q2[Y].  E<> (observer.received && observer.x == 6)
q3[N].  E<> (observer.received && observer.x < 6)
q4[Y].  E<> (observer.received && observer.x == 182)
q5[N].  E<> (observer.received && observer.x > 182)
```

Note that in each query, `[Y]` or `[N]` indicates the checking result of the query as satisfied or not satisfied. The result of `q1` shows that the frame transmission time of `msg1` is 5MT, which exactly matches the setting of the length of *msg1*. The results of `q2` and `q3` show that the best case response time (BCRT) of `msg1` is 6MT, which is the sum of the frame length (5MT) and the static slot offset (1MT). The results of `q4` and `q5` show that the best case response time (WCRT) of `msg1` is 182MT, which is the sum of the lengths of the static segment (30MT), the dynamic segment (150MT) and the NIT (2MT).

Discussions. In SR1, we focus on frame transmissions in *FlexRay model* and checked some related properties. With the results, we conclude that the framework is built right in the scope of normal frame transmissions of the FlexRay protocol. With only a few properties checked, one may doubt that it is not sufficient to confirm that *FlexRay model* of the framework conform to the specifications of the FlexRay protocol. For this issue, we argue that since we only focus on normal frame transmissions of the FlexRay protocol, the checking results are satisfiable at current status of the framework. Furthermore, since the structure of the framework follows the structure of the specifications of the FlexRay protocol, when we want to extend the functionalities of *FlexRay model*, current status of *FlexRay model* can be a base to implement the extensions.

In SR2, the checking of the response times shows that the framework is able to check timing properties with the help of *Observer*. This is a common technique for checking complex properties since UPPAAL only support simple timed computational tree logic (TCTL) formulas. Also, to decide the value of BCET and WCET to be filled in a query, currently we have to guess according to the parameters of a system. This may result some trial and error, or we may utilize some traditional timing analysis techniques.

For the feasibility of applying the framework in the industry, since in this paper we only focus on building *FlexRay model*, the modeling of the tasks in a system is left to developers. Therefore, developers have to be familiar with the usage of UPPAAL. Also, it is necessary to have a methodology of modeling tasks, which may be adopted from the experiences of the modeling on integrated platforms. Another issue is the performance of the framework. From the results shown in Table 1, the state space is quite large considering that the system of SR1 is very simple. The performance issue would be a major problem when applying the framework to industrial automotive system designs.

6 Conclusion and Future Work

In this paper, an UPPAAL framework for model checking automotive system designs with FlexRay protocol is introduced and evaluated. The framework

consists of FlexRay model and application model: the former is built by abstractions to the FlexRay protocol and can be reused for different applications with proper parameter settings represented by global variables in UPPAAL. To the best of our knowledge, the framework is the first attempt for model checking automotive system designs considering communication protocols. To evaluate the framework, we demonstrated two simple systems and checked some queries/properties. From the results, we conclude that the framework is built right in accordance with normal frame transmissions of the FlexRay protocol and is able to check timing properties.

In this paper, we showed that a reusable module on top of UPPAAL, i.e. FlexRay model, could be realized for verification of applications with FlexRay protocol. We argue that only providing a general purpose model checker is not sufficient for verifying practical systems. Additional descriptions and mechanisms such as scheduling of tasks and emulation of hardware devices are usually needed to precisely model and verify the behavior of practical systems. Furthermore, these additional mechanisms are usually common for systems belonging to a specific application domain and are possible to be provided as reusable frameworks and libraries. Therefore, integrating such frameworks and libraries is crucial for promoting practical applications of model checkers in the industry.

Currently, the framework can only support scheduling analysis in systems that an ECU has only one task, or developers have to build scheduler modules, which is not easy. Therefore, we plan to add scheduler modules to support general scheduling analysis. We also plan to conduct more experiments on practical automotive systems to discover more usages and possible improvements of the framework.

References

1. Altran Technologies: FlexRay Specifications Version 3.0.1 (2010)
2. Bel Mokadem, H., Berard, B., Gourcuff, V., De Smet, O., Roussel, J.-M.: Verification of a timed multitask system with UPPAAL. IEEE Trans. Autom. Sci. Eng. **7**(4), 921–932 (2010)
3. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. Hybrid Systems III. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)
4. Bøgholm, T., Kragh-Hansen, H., Olsen, P., Thomsen, B., Larsen, K.G.: Model-based schedulability analysis of safety critical hard real-time java programs. In: Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'08), pp. 106–114 (2008)
5. David, A., Rasmussen, J.I., Larsen, K.G., Skou, A.: Model-based framework for schedulability analysis using Uppaal 4.1. Model-Based Design for Embedded Systems. Computational Analysis, Synthesis, and Design of Dynamic Systems, pp. 93–119. CRC Press, Boca Raton (2009)
6. Gerke, M., Ehlers, R., Finkbeiner, B., Peter, H.-J.: Model checking the FlexRay physical layer protocol. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 132–147. Springer, Heidelberg (2010)

7. Giusto, P., Ferrari, A., Lavagno, L., Brunel, J.Y., Fourgeau, E., Sangiovanni-Vincentelli, A.: Automotive virtual integration platforms: why's, what's, and how's. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 370–378 (2002)
8. Hagiescu, A., Bordoloi, U.D., Chakraborty, S., Sampath, P., Ganesan, P.V.V., Ramesh, S.: Performance analysis of FlexRay-based ECU networks. In: DAC'07, pp. 284–289 (2007)
9. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008)
10. Hiraoka, T., Eto, S., Nishihara, O., Kumamoto, H.: Fault tolerant design for X-by-wire vehicle. In: SICE'04 Annual Conference, vol. 3, pp. 1940–1945 (2004)
11. Jung, K.H., Song, M.G., Lee, D.I., Jin, S.H.: Priority-based scheduling of dynamic segment in FlexRay network. In: International Conference on Control, Automation and Systems (ICCAS'08), pp. 1036–1041 (2008)
12. Malinský, J., Novák, J.: Verification of FlexRay start-up mechanism by timed automata. Metrol. Measur. Syst. **17**(3), 461–480 (2010)
13. Navet, N., Song, Y., Simonot-Lion, F., Wilwert, C.: Trends in automotive communication systems. Proc. IEEE **93**(6), 1204–1223 (2005)
14. Qtronic GmbH, Germany: Virtual integration and test of automotive ECUs. In: Automotive Testing Expo North America, ASAM Open Technology Forum (2011)
15. Sangiovanni-Vincentelli, A.: Electronic-system design in the automobile industry. IEEE Micro **23**(3), 8–18 (2003)
16. Tanasa, B., Bordoloi, U., Kosuch, S., Eles, P., Peng, Z.: Schedulability analysis for the dynamic segment of FlexRay: a generalization to slot multiplexing. In: IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS'12), pp. 185–194 (2012)
17. Zeng, H., Ghosal, A., Di Natale, M.: Timing analysis and optimization of FlexRay dynamic segment. In: IEEE 10th International Conference on Computer and Information Technology (CIT'10), pp. 1932–1939 (2010)
18. UPPAAL models used in this paper: https://github.com/h-lin/FTSCS2013