Formal Verification of Multifunction Vehicle Bus

Lianyi Zhang[1,2] Duzheng Qing[1,2] Lixin Yu[2] Mo Xia[3] Han Zhang[2] Zhiping Li[2]

[1] Science and Technology on Special System Simulation Laboratory, Beijing Simulation Center, China
[2] State Key Laboratory of Intelligent Manufacturing System Technology, Beijing Institude of Electronic System Engineering, China
[3] Software School, Tsinghua University, Beijing, China

Abstract

Multifunction Vehicle Bus (MVB) is a critical component in the Train Communication Network (TCN), which represents a challenging problem for model checking. Although model checking is widely used in circuit and software verification, it is hardly for verification of the MVB or TCN, in terms of modelling of MVB components and making appropriate specification. The study described in this paper aims at evaluating and experimenting the industrial application of verification by model checking, and provides a complete system modelling and specification describing technique. The model of MVB consists of device model, communication model and specification model translated from LSCs, a scenario description. Experiments results with SPIN checking tool illustrate effectiveness of our approach.

Keywords: vehicular communication, protocol verification, model checking, specification

1 Introduction

Multifunction Vehicle Bus (MVB) in Train Communication Network (TCN) is widely used in most of the modern train control techniques of transportation software system. MVBs change roles of master or slaves under mastership transfer protocol in IEC standards (Kirrmann and Zuber, 2001). How to ensure security of an embedded vehicle control software system which implements the function and protocol has become an important issue.

The traditional method to verify MVB uses simulation technique (Jiménez et al., 2006) or test methods (Zhiwu et al., 2008). These approach need semi-finished MVB devices and programs, error-tolerance decode algorithm and samples of transmitted data among devices. However, simulation and test cannot provide completeness verifica-tion.

Model checking (Queille and Sifakis,1982; Clarke et al., 1999) is a method for automatically verifying finite state systems, using an exhaustive search of the state of a system model to determine whether a specification is satisfied or violated, which has been applied in circuit and software verification. Although model checking is widely used, it is hardly for verification of the MVB or TCN, in terms of modelling of MVB components and making appropriate specification. System modelling prefers fewer states, avoiding states space explosion, and specification making requires different properties synthesis with light weight manual work.

This paper presents a modelling approach for MVB based on finite state model checking and a specification generated method. The remainder of this paper is organized as follows. The following section gives background details of MVB, mastership transfer protocol and preliminaries model checking with temporal logic. Section 3 propose a modelling method for system component and communication with process and finite state automata. Section 4 show property generated and modelling method compound with previous system model. Experiment results in Section 5 demonstrate our methods and Section 6 make conclusion.

2 Background

In this section, we first introduce the MVB and the master transfer protocol. Then model checking with temporal logic is reviewed.

2.1 MVB and Master Transfer

2.1.1 Multifunction Vehicle Bus

The on-board train communication system has been widely used for modern railways. The MVB is a component of the TCN which is used in most of the modern train control systems. The TCN has been defined by the IEC(international Electrotechnical Commission); it is the Vehicle Bus specified to connect standard equipment. It provides both the interconnection of programmable equipment pieces amongst themselves and the connection of this equipment with its sensors and actors. It can also be used as a Train Bus in trains which are not separated during normal operation.

The MVB defines two types of devices: Master and Slave. Each Vehicle Bus and Train Bus has one Master node and several Slave ones. The Master sends information, the *Master_Frame*, to a number of Slave devices. The Slave receives information from the Bus and sends information, the *Slave_Frame*, in response to the Master. A Master_Frame and the corresponding Slave_Frame form a telegram. All devices decode the Master_Frame. The addressed source device then replies with its Slave_Frame, which may be received by several other devices.

2.1.2 Mastership Transfer

Since a single Master presents a single point of failure, mastership may be assumed by several Bus Administrators(BAs), one at a time. To increase availability, mastership can be shared by two or more BAs, which both take charge of mastership for the duration of a turn. Mastership is transferred from BusAdmin to BusAdmin within a few milliseconds in case of failure. To exercise redundancy, mastership is transferred every few seconds by a token frame. Consequently, all BAs are organized in a logical ring. A token passing mechanism ensures that only one BusAdmin become Master.

In the IEC 61375-1 international standard, Mastership Transfer describes the protocol which selects a Master form one of several BAs and ensures Mastership Transfer at the end of a turn or upon the occurrence of a failure. A token passing algorithm is defined in the IEC standard to ensure a round-robin access of all BAs to the Bus:

1. after the loss of the Master, staggering of the timeouts ensures that only one of the BAs become Master;
2. a Master exercise mastership for the duration of one turn;
3. after its turn, the Master looks for the next BusAdmin and reads its Device Status, which indicates if this device is a configured BusAdmin;
4. a Master may only pass mastership to a configured and actualized BusAdmin;
5. if the device is not a configured and actualized BusAdmin, the Master looks for the next BusAdmin after the next turn;
6. if the device is a configured and actualized BA, the Master offers mastership to it by sending a Mastership Transfer Request;
7. if the device accepts mastership in its Mastership Transfer Response, or if no answer comes, the Master retires to become a standby Master and monitors the Bus traffic for mastership offer or Bus silence;
8. if the other device rejects mastership, the current Master retains mastership for one more turn, after which the Master tries the next device in its BAs list;
9. a standby BusAdmin becomes Master if it accepts a Mastership Transfer Request or if it detects no Bus activity during a time greater than a defined timeout;

2.2 Model Checking with temporal logic

For model checking system against some specification, we need both system model structure and description of property. Communicating Finite State Machines (CFSMs) are natural models for systems of concurrently running process, especially asynchronous reactive system. The concurrency of process is captured at the semantics level of CFSMs by the interleaving of processes executions. Process exchange messages between each other asynchronously over a set of message channels, which are interpreted at the semantic level as unbounded FIFO message queues. A sender process continues its local execution after sending a message to a channel, and a receiver process is blocked when it tries to receive a message that is not available in the respective channel.

Definition 1 *(Communicating Finite State Machines). A system of communicating finite state machines (CFSM) is a tuple* $\mathscr{S} = (P,V,M,C,succ)$*, where*

- *P is a finite set of* process*. Each process* p_i *is a pair* (S_i, s_0^i) *where* S_i *is a finite set of states of* p_i *and* $s_0^i \in S_i$ *is the initial state. For any two different process* p_i *and* p_j*, we put the restriction that* $S_i \cup S_j =$*, i.e., their sets of states are disjoint.*
- *V is a finite set of* variable*, may be global or local in process.*
- *M is a finite set of* message symbols*.*
- *C is finite of* messages channels*. Each channel is associated with a subset of message symbols* $M' \subset M$ *such that only the messages in* M' *can be exchanged in the buffer. Moreover, for each channel* $c \in C$ *and each message symbol m in the subset of M associated with c, we call* (c,m) *a message type.*
- *succ is a finite set of* local transitions (s,e,s') *where s and* s' *are states of some same process* p_i*, and e is* (g,u)*: g is* guard condition *consist of both boolean formula with or without a message passing event in the form b*!*m or b*?*m such that (1)*$c \in C$ *and (2)*$m \in M$ *can be exchanged in the buffer c; u is updates of variables as* $v' = u(v)$*.*

The semantics of CFSMs is defined using the concepts of configurations and reachability.

Definition 2 *(Configuration). Given a CFSM system* $\mathscr{S} = (P,V,M,C,succ)$*, a* configuration*(or* global state*) of the system is a tuple* $(s^1,...,s^{|P|},v,q^1,...,q^{|C|})$ *such that*

- *each* s^i *is a state of the process* p_i*, and*
- *each* q^i *is a queue of messages exchangeable in the channel* c_i*, and*
- *v is a valuation of all variables.*

Consider two configuration s_1 *and* s_2*, let* $s_1 = (s_1^1,...,s_1^{|P|}, v_1, q_1^1,...,q_1^{|C|})$ *and* $s_2 = (s_2^1,...,s_2^{|P|}, v_2, q_2^1,...,q_2^{|C|})$*. We define that* s_2 *is a successor of* s_1*, denoted by* $s_1 \Rightarrow s_2$*, if the following is satisfied:*

- *There exists a process $p_i \in P$ such that (1) for all $j \neq i$ we have that $s_1^j = s_2^j$; and (2)$(s_1^i, e, s_2^i) \in succ$.*
- $(v_2, q_2^1, ..., q_2^{|C|}) = post_e(v_1, q_1^1, ..., q_1^{|C|})$, *where $post_e$ update the valuation of variables from v_1 to v_2 and change message queues contents of respective channels.*

After modelling system, we should describe specification under which the model to be verified. Temporal logic, such as Linear Temporal Logic, which extends proposition logic with temporal operator, is a good choice of description of system properties.

Definition 3 *(Linear Temperoal Logic). Linear Temporal Logic(LTL) has the following syntax given in Backus Naur form:*

$$\phi ::= \bot \mid \top \mid p \mid (p) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid X\phi \mid F\phi \mid G\phi \mid \phi U \psi \mid \phi W \psi \mid \phi R \psi \quad (1)$$

where p is any proposition atom from sone set of atoms. Temporal operator X means next state, F means some future state, and G means all states. The next three, U, W and R, are called Until, Release, *and* Weak-until, *respectively.*

According to the requirement specified in the standard, suppose that there are n BAs altogether, the Mastership Transfer must satisfy the following properties:

1. there cannot be more than one Master at one time; it is written in the LTL as shown in

$$G\neg(BA\_Master_i \wedge BA\_Master_j), i, j = 1, ..., n, i \neq j$$

2. there cannot be no Master at one time; it is written in the LTL as shown in

$$G\neg(\bigwedge_{i=1}^{n} BA\_Standby_i).$$

3 System Modelling

MVBs under Mastership Transfer protocol has two main component to be modelled. In this section, we present our modelling approach for both BusAdmins and the communication mechanism amongst BAs.

3.1 Bus Administrator Modelling

3.1.1 basic data structure

First we define the data structure of Bus Administrator and message frame exchanged amongst different BAs. Bus Administrator data structure defines local variables used for BA itself, including different timers, flag, address, etc. Most important fields of BA structure are *send* and *recv* buffer channel for communicating. These channel contains buffer blocks, each of that consists of both message frame and channel pointer for convenience.

Listing 1. Bus Admin structure

```
typedef BA_DEF
{
   byte T_standby; /*t_standby Timer*/
   byte Turn;  /*turn Timer*/
   bit T_find_next; /*t_find_next Timer*/
   bit T_interim; /*t_interim Timer*/
   bit ACT;    /*BA actualized state flag*/
   byte rank;  /*BA sequence number in BA-
       list*/
   byte adr;   /*BA address in memory*/
   chan send = [10] of {FRAME, chan};
      /*BA sender buffer channel*/
   chan recv = [10] of {FRAME, chan};
      /*BA receiver buffer channel*/
}
```

Listing 2. enumeration of BA state

```
mtype =
{ ba_STANDBY_MASTER,
  ba_REGULAR_MASTER,
  ba_END_OF_TURN,
  ba_FIND_NEXT,
  ba_INTERIM_MASTER}
```

Enumeration of BA state consists of all major distinguished states of BA under Mastership Transfer protocol.

Listing 3. message frame structure

```
typedef FRAME
{
   mtype type; /*Frame type*/
   bit  data; /*Frame data*/
   byte from; /*Source BA ID of frame*/
   byte to; /*Target BA ID of frame*/
}
```

Message frame structure defines type, data, and source and target BA ID of messages frames.

Listing 4. enumeration of message frame type

```
mtype =
{ MASTERSHIP_OFFERED,
  MASTERSHIP_RESPONSE,
  STATUS_REQUEST,
  STATUS_RESPONSE,
  REGULAR}
```

Enumeration of message type consists of all major distinguished states of BA under Mastership Transfer protocol.

3.1.2 finite state machine of Bus Administrator

As defined by CFSM in previous section, our model consists of several concurrency running processes, each of that is an instance of process type BA_FSM, which is identified by argument ID. Each process represent an executing Bus Administrator with a member index by same ID in a BA_DEF structure array.

BA_FSM constructs a finite state machine of Bus Administrator. In each state, BA exercise respective operations, receive or sends message frames and transit to other state on conditions.

Listing 5. BA_FSM process structure

```
proctype BA_FSM(byte ID)
{
    STANDBY_MASTER: /*standby state*/
    ...
    /*acts no master operation: if it
        accepts a Mastership Transfer
        Request or if it detect no Bus
        activity during a time greater than
        t_standby, goto REGULAR_MASTER*/
    REGULAR_MASTER: /*master exercise state
        */
    ...
    /*acts master operation: if it detects
        master conflict, goto STANDBY_MASTER
        ; or if it ends its current turn,
        goto END_OF_TURN*/
    END_OF_TURN: /*end of master turn*/
    ...
    /*looks for next BA in BA-list and send
        Status Request; if BA-list exhausts,
         goto REGULAR_MASTER*/
    FIND_NEXT:  /*find next BA to be master
        */
    ...
    /*gets the Status Response: if the BA is
         configured and actualized, then
        offers mastership to it by sending
        Mastership Transfer Request and goto
         INTERIM_MASTER; else if the BA is
        not actualized or time-out without
        response, goto END_OF_TURN*/
    INTERIM_MASTER: /*interim master state*/
    ...
    /*gets the Mastership Transfer Response
        and goto STANDBY_MASTER. Meanwhile,
        if the response is non-acceptance or
         time-out without response, report
        error*/
}
```

3.2 Communication and Timing Modelling

3.2.1 communication mechanism of BusAdmin

Instead of modelling real Bus component, which may introduce more complex concurrency process, we realize communication between BusAdmins by channels. In initial model setting, each BA has sender and receiver channels both with buffer capacity of 10.

asynchronous communication with buffers By positive capacity buffered channel, BAs communicates with each other asynchronously. These messages receiving and sending actions are similar to pulling and posting mails through intermediary.

- when BA receives(pull) message from *recv* channel, it execute

```
BA[ID].recv ? (temp_frame, BA[ID].
    send)
```

  to get message frame into temporal frame to be parsed; meanwhile, get sender's receive channel pointer as send channel of itself, prepared to be used in following sending actions.

- when BA send(post) message to *send* channel, it execute

```
BA[ID].send ! (temp_frame, BA[ID].
    recv)
```

  to put temporal frame into previous channel pointer(another BA's recv channel); meanwhile, send it's recv channel as channel pointer to be received by the target receiver.

In above communication realization, we make channel as part of communication content, reduce amount of channels used in whole model and alleviate state explorations of states caused by channel numbers.

non-blocking communication When channel is empty or full, respectively, receiving or sending processes block at previous execution statement. Obviously, it is inflexible for modelling more complex system execution situation. So we realized non-blocking communication use *full*, *nfull* and *atomic* primitives.

When sender tries to send, it first confirm whether send channel is full or not, and exclusively satisfies either $full$ or $nfull$ guard condition. If send channel is full, sender process skips sending and execute next statement. If not, sender sends message successfully.

```
do
::full(BA[ID].send)->skip;
::nfull(BA[ID].send)->
   BA[ID].send!(temp_frame,BA[ID].recv);
od;
```

Situation is slightly different for receiver process. We encapsulate receiving execution as atomic segment and provides *else* clause besides. When receiver cannot receive message frame from recv channel as the channel is empty, it will chose *else* bypath and will not be blocked.

```
do
::atomic
   {BA[ID].recv?(temp_frame,BA[ID].send)->
      ...
   }
::else->...
od;
```

3.2.2 timing mechanism

In Mastership Transfer protocol, there are many local timer in each process, such as $T\_standby$, $Turn$, etc. As CFSMs semantics is asynchronous models, different timer in process need to be synchronised.

Many modelling language has synchronous channels, which is different from asynchronous channels, the former can be treated as zero-capacity channel that requires sender and receiver must communicated at the same instant.

Based on the synchronous channel structure, we can make timer in different process has same steps. We model another timing manager process to coordinate different timers. Timer is not updated in each process, but be sent to timing manager by each process before updates through synchronous communication. Then timing manage updates all timer counts in atomic segment and synchronously sends timer back to each process. By timing manager, we realize timers in different process to step by same rate.

4 Property Modelling

After we get system model, we need make specification to be verified. In this section, we first introduce properties classification about our model, then for handling troubles of complex property description and gaining benefits of synthesis, we introduce live sequence chart and combine it with previous system model.

4.1 Property Classification

4.1.1 safety property

A safety property states that some bad thing never happens, representing requirements that should be continuously maintained by the system. To our models, we have following safety property to be verified:

- BusAdmin 0 (BA[0]) and BusAdmin 1 (BA[1]) cannot be masters at same time:

$$G\ mutex \tag{2}$$

where

```
#define mutex
(!(ba0_cur==ba_REGULAR_MASTER &&
    ba1_cur==ba_REGULAR_MASTER))
```

- Once BA[0] becomes master, then system must has a master forever:

$$G(\ ba0\_master \rightarrow G\ !nonemaster)) \tag{3}$$

where

```
#define ba0_master
(ba0_cur==ba_REGULAR_MASTER)
#define nonemaster
(ba0_cur==ba_STANDBY_MASTER &&
 ba1_cur==ba_STANDBY_MASTER &&
 ba2_cur==ba_STANDBY_MASTER)
```

4.1.2 liveness property

A liveness property states that some good thing eventually happens, representing requirements whose eventual realization must be ensured. To our models, we have following liveness property to be verified:

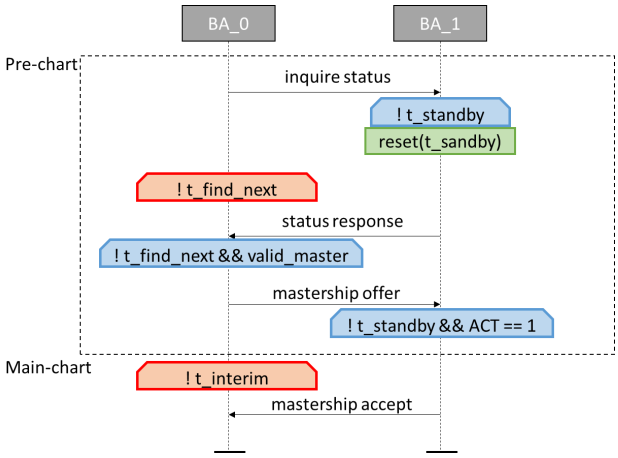

Figure 1. An example of LSC specification: *BA*_0 transfer mas-tership to *BA*_1.

- It is required that BA[0], BA[1] and BA[2] all can infinitely often be the master.

$$GF\ ba0\_master\ \&\&\ GF\ ba1\_master \\ \&\&\ GF\ ba2\_master \tag{4}$$

where $ba1\_master$ and $ba2\_master$ is defined similar as $ba0\_master$.

- After BA[0] becomes the master, it can eventually relieve mastership.

$$G\ (ba0\_master \rightarrow F\ ba0\_standby) \tag{5}$$

where $ba0\_standby$ is defined similar as $ba0\_master$.

4.2 Live sequence charts

Live Sequence Chart (LSC) (Kugler and Segall, 2009; Li et al., 2010) use instance lines, expressions, messages, conditions and updates to describe scenario of interactions among sequence processes. We can iteratively refine spec-ification described by LSCs, that is introduced in initial system design stage and reused for the whole software de-veloping procedures.

Example 1 *In Figure.1, an Live Sequence Chart describes a fragment procedure of mastership transfer from BA_0 to BA_1, which starts from BA_0 send inquire status mes-sage and ends with BA_1 sends mastership accept mes-sage. Red hexagon represents hot condition, which can-not be violated. Blue hexagon represents cool condition, which once be violated, then go back to top and make a new execution.*

Given formal syntax and sematic of LSCs, we can combine normal specification described by LSC with existing formal logics and model checking to verify the properties provided by developer and stakeholder.

Main process of LSC described scenario-based system properties formal description and model checking is that,

1. Use normal LSC to describe process instance interaction scenario in system model, identify the Pre-chart and Main-chart.

2. Translate LSC to Observer Automata *ObsA*. *ObsA* monitors these messages communication, conditions satisfaction and updates consistency proposed by LSC.

3. Combine *ObsA* with original system model $\mathscr{S}$, without introducing side effects in original system operation.

4. Formally, check property

$$G\ (ObsA.l_{min} \rightarrow F\ ObsA.l_{max})$$

where $ObsA.l_{min}$ identifies top of Main-chart, $ObsA.l_{max}$ identities bottom of Main-chart. This property represents that each time the system enters the main chart of LSC, it will eventually reach the bottom of the main chart. Then LSC scenario property is reduced to a classical real-time model checking problem.

4.3 Observer Automata modelling

To translate LSC to observer automata, we need partition LSC into locations, simregions, and cuts firstly. Then a run of observer automata is a successive transitions among states space consisting of cuts and variables valuations.

One requirement for observer automata is that it can monitor concerned information of interactions and configurations in LSC, such as messages. In addition, observer automata need also capture other information such as timers. Meanwhile, observer cannot bring side effects that disturb normal executions of original system model.

We use copy acceptance communication, which pulls a copy message from channel and has no effect on channel information and structure, like

```
BA[ID].recv ? <temp_frame, temp_chan>
```

where syntactic sugar <> means copy message from *recv* to *temp_fram* and *temp_chan*. As mentioned before, timers implement is similar to messages communication, so it is also monitored by observer automata as reference copy.

We define boolean (bit) variables *obsAlmin* and *obsAlmax* as flag denotation for $ObsA.l_{min}$ and $ObsA.l_{max}$ respectively, with initial value of $false(0)$. Based on the above, a monitor process which realizes observer automata *ObsA* and a LSC specification can be both added to our model checking of MVB protocol.

5 Experiments

We implement our approach by explicit states model checking tool SPIN (Holzmann,1997). Its modelling language Promela has all the model elements we proposed previously. To check LSC specification, we implement a tools translate graphic LSC to modelling codes. We run our model checking on 32-bit Window7 platform with 2GB RAM memory limitation. To exhibit the use of our modelling and property setting works

We show the experiments results of MVB protocol model checking against 5 property presented previously in this paper in Table.1.

- First two lines indicates results of first two safety property checking. As safety property is violated by finite prefix of execution, the search depth is equivalent to counterexample depth.

- Lines 3, 4 indicate results of two liveness property checking. Different from above safety property, a counterexample trace that violates liveness property is a infinite suffix of executions, which consists of a lasso structure. Model checking algorithm for search of such lasso is a double-DFS algorithm using stack and has tables. Consequently, these counterexamples depth are less than checking depth reached.

- Last line is result of checking translated LSC property. Obviously, checking a property described by LSC and translated into observer automata is much harder than simple safety and liveness property. It is resulted both from LSC's rich expression and observer automata complex monitoring functions. In this experiment, we can not find counterexample before memory exhausts. Conservatively speaking, the specification described by LSC is satisfied by MVB protocol.

6 Conclusions

Model checking of Multiple Vehicle Bus under mastership transfer protocol shows an industrial verification case study, which combines both device model, communication model and specification model in a unified modelling framework. One benefit of our approach is that, we model Bus administrator as modules, and use channels efficiently. It is an immense improvement on modelling technique compared to (Xia et al., 2013). Another benefit is that we introduce LSCs to make specification of more complex process interaction scenarios. Moreover, with help of translating LSC into observer automata and verifying automata location based liveness property, we can confirm whether LSCs specification is satisfied or not.

We think we have two aspects for future work. One is further improvement of our modelling works and translation from specification to automata. Because the size of models directly affect checking efficiency, we expect smaller models and make possible abstraction. The other extending maybe introduce synthesis technique, which starts from specifications, and seek possible system models that satisfies all the inferred properties.

Table 1. Experiments results of MVB protocol.

Property ID	Transitions	Atomic steps	Memory usage (MB)	Depth reached	Counterexample depth
1	399	127	2.391	966	966
2	347	112	2.391	846	846
3	574767	183018	15.379	1999	1903
4	2964	112	2.586	978	940
5	12201602	3032340	2024.453	1999	—

Acknowledgement. This paper is supported in part by the National Key R&D Program of Chian No. 2017YFC0820100.

References

Edmund M Clarke, Orna Grumberg, and Doron A Peled. *Model checking*. MIT press, 1999.

Gerard J Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

Jaime Jiménez, Iker Hoyos, Carlos Cuadrado, Jon Andreu, and Aitzol Zuloaga. Simulation of message data in a testbench for the multifunction vehicle bus. In *IEEE Industrial Electronics, IECON 2006-32nd Annual Conference on*, pages 4666–4671. IEEE, 2006.

Hubert Kirrmann and Pierre A Zuber. The iec/ieee train communication network. *Micro, IEEE*, 21(2):81–92, 2001.

Hillel Kugler and Itai Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 77–91. Springer, 2009.

Shuhao Li, Sandie Balaguer, Alexandre David, Kim G Larsen, Brian Nielsen, and Saulius Pusinskas. Scenario-based verification of real-time systems using Uppaal. *Formal Methods in System Design*, 37(2-3):200–264, 2010.

Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

Mo Xia, Kueiming Lo, Shuangjia Shao, and Mian Sun. Formal modeling and verification for MVB. *Journal of Applied Mathematics*, 2013, 2013.

Huang Zhiwu, Zhou Sheng, Gui Weihua, and Liu Jianfeng. Research and design of protocol analyzer for multifunction vehicle bus. In *Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on*, pages 8358–8361. IEEE, 2008.