Formal Correctness of an Automotive Bus Controller Implementation at Gate-Level

Eyad Alkassar, Peter Böhm, and Steffen Knapp

Abstract We formalize the correctness of a real-time scheduler in a time-triggered architecture. Where previous research elaborated on real-time protocol correctness, we extend this work to gate-level hardware. This requires a sophisticated analysis of analog bit-level synchronization and message transmission. Our case-study is a concrete automotive bus controller (ABC). For a set of interconnected ABCs we formally prove at gate-level, that all ABCs are synchronized tight enough such that messages are broadcast correctly. Proofs have been carried out in the interactive theorem prover Isabelle/HOL using the NuSMV model checker. To the best of our knowledge, this is the first effort formally tackling scheduler correctness at gate-level.

1 Introduction

As more and more safety-critical functions in modern automobiles are controlled by embedded computer systems, formal verification emerges as the only technique to ensure the demanded degree of reliability. When analyzing correctness, as a bottom layer, often, only some synchronous model of distributed electronic control units (ECUs) sharing messages in lock-step is assumed. However, such models are im-

Eyad Alkassar[1] · Steffen Knapp[1]
Saarland University, Dept. of Computer Science, 66123 Saarbrücken, Germany
e-mail: {eyad,sknapp}@wjpserver.cs.uni-sb.de

Peter Böhm[1]
Oxford University Computing Laboratory, Wolfson Building, Oxford, OX1 3QD, England
e-mail: peter.boehm@comlab.ox.ac.uk

[1] Work partially funded by the German Research Foundation (DFG), by the German Federal Ministry of Education and Research (BMBF), and by the International Max Planck Research School (IMPRS).

Please use the following format when citing this chapter:

Alkassar, E., Böhm, P. and Knapp, S., 2008, in IFIP International Federation for Information Processing, Volume 271; *Distributed Embedded Systems: Design, Middleware and Resources*; Bernd Kleinjohann, Lisa Kleinjohann, Wayne Wolf; (Boston: Springer), pp. 57–67.

plemented at gate-level as highly asynchronous time-triggered systems. Hence it can not suffice to verify certain aspects of a system, as algorithms or protocols only.

In this paper we examine a distributed system implementation consisting of ECUs connected by a bus. Our study has to combine arguments from three different areas: (i) asynchronous bit-level transmission, (ii) scheduling correctness, and (iii) classical digital hardware verification at gate-level.

Our contribution is to show, by an extended case-study, how analog, real-time and digital proofs can be integrated into one pervasive correctness statement.

The hardware model has been formalized in the Isabelle/HOL theorem prover [11] based on boolean gates. It can be translated to Verilog and run on a FPGA. All lemmata relating to scheduling correctness have been formally proven in Isabelle/HOL. We have made heavy use of the model checker NuSMV [5] and automatic tools, e.g. IHaVeIt [18], especially for the purely digital lemmata. Most lemmata dealing with analog communication (formalized using reals) have been shown interactively.

Overview. The correctness of our gate-level implementation splits in two main parts: (i) the correctness of the transmission of single messages and (ii) the correctness of the scheduling mechanism initiating the message transmission and providing a common time base. Next we outline these two verification goals in detail.

The verification of asynchronous communication systems must, at some point, deal with the low-level bit transmission between two ECUs connected to the same bus. The core idea is to ensure that the value broadcast on the bus is stable long enough such that it can be sampled correctly by the receiver. To stay within such a so-called sampling window, the local clocks on the ECUs should not drift apart more than a few clock ticks and therefore need to be synchronized regularly. This is achieved by a message encoding that enforces the broadcast of special bit sequences to be used for synchronization. The correctness of this low-level transmission mechanism cannot be carried out in a digital, synchronous model. It involves asynchronous and real-time-triggered register models taking setup and hold-times of registers as well as metastability into account. Our efforts in this respect are based on [3, 8, 16].

Ensuring correct message transmission between two ECUs is only a part of the overall correctness. Let us consider a set of interconnected ECUs. The scheduler has to avoid bus contention, i.e. to ensure that only one ECU is allowed to broadcast at a time and that all others are only listening. For that, time is divided into rounds, which are further subdivided into slots. A fixed schedule assigns a unique sender to a given slot number. The gate-level implementation of the scheduler has to ensure that all ECUs have roughly the same notion of the slot-start and end times, i.e. they must agree on the current sender and the transmission interval. Due to drifting clocks some synchronization algorithm becomes necessary. We use a simple idea: A cycle offset is added at the beginning and end of each slot. This offset is chosen large enough to compensate the maximal clock drift that can occur during a full round. The local timers are synchronized only once, at the beginning of each round. This is done by choosing a distinguished master ECU, being the first sender in a round.

The combination of the results into a lock-step and synchronous view of the system is now simple. The scheduler correctness ensures that always only one ECU is sending and all other ECUs do listen. Then we can conclude from the first part that the broadcast data is correctly received by all ECUs.

Organization of the paper: In the remainder of this section we discuss the related work. In Section 2 we introduce our ABC implementation. Our verification approach is detailed in Section 3. Finally we conclude in Section 4.

Related Work. Serial interfaces were subject to formal verification in the work of Berry *et al.* [1]. They specified a UART model in a synchronous language and proved a set of safety properties regarding FIFO queues. Based on that a hardware description can be generated and run on a FPGA. However, data transmission was not analyzed.

A recent proof of the Biphase-Mark protocol has been proposed by Brown and Pike [4]. Their models include metastability but verification is only done at specification level, rather than at the concrete hardware. The models were extracted manually.

Formal verification of clock synchronization in timed systems has a long history [9, 12, 17]. Almost all approaches focused on algorithmic correctness, rather than on concrete system or even hardware verification. As an exception Bevier and Young [2] describe the verification of a low-level hardware implementation of the Oral Message algorithm. The presented hardware model is quite simplified, as synchronous data transmission is assumed.

Formal proofs of a clock-synchronization circuit were reported by Miner [10]. Based on abstract state machines, a correctness proof of a variant of the Welch-Lynch algorithm was carried out in PVS. However, the algorithm is only manually translated to a hardware specification, which is finally refined semi-automatically to a gate-level implementation. No formal link between both is reported. Besides, low-level bit transmission is not covered in the formal reasoning.

The formal analysis of large bus architectures was tackled among others by Rushby [15] and Zhang [19]. Rushby worked on the time-triggered-architecture (TTA), and showed correctness of several key algorithms as group membership and clock synchronization. Assuming correct clock synchronization, Zhang verified properties of the Flexray bus guardian. Both approaches do not deal with any hardware implementation. The respective standard is translated to a formal specification by hand.

In [14] Rushby proposes the separation of the verification of timing-related properties (as clock synchronization) and protocol specifications. A set of requirements is identified, which an implementation of a scheduler (e.g. in hardware) has to obey. In short (i) clock synchronization and (ii) a round offset large enough to compensate the maximum clock drift are assumed. The central result is a formal and generic PVS simulation proof between the real-time system and its lock-step and synchronous specification. Whereas the required assumptions are similar to ours, they have not been discharged for concrete hardware.

In [12] Rushby's framework is instantiated with the time triggered protocol (TTP). Pike [13] corrects and extends Rushby's work, and instantiates the new framework with SPIDER, a *fly-by-wire* communication bus used by NASA. The time-triggered model was extracted from the hardware design by hand. But neither approaches proved correctness of any gate-level hardware.

2 Automotive Bus Controller (ABC) Implementation

We consider a time-triggered scenario. Time is divided into so-called rounds each consisting of *ns* slots. We uniquely identify slots by a tuple consisting of a round-number $r \in \mathbb{N}$ and a slot-number $s \in [0 : ns-1]$. Predecessors $(r,s)-1$ and successors $(r,s)+1$ are computed modulo *ns*.

The ABC is split in four main parts: (a) the host-interface provides the connection to the host, e.g. a microprocessor, and contains configuration registers (b) the send-environment performs the actual message broadcast and contains a send-buffer (c) the receive-environment takes care of the message reception and contains a receive-buffer (d) the schedule-environment is responsible for the clock synchronization and the obedience to the schedule.

Configuration Parameter. Unless synchronization is performed, slots are locally *T* hardware cycles long. A slot can be further subdivided into three parts; an initial as well as a final offset (each *off* hardware cycles) and a transmission window (*tc* hardware cycles). The length of the transmission window is implicitly given by the slot-length and the offset. Within each slot a fixed-length message of ℓ bytes is broadcast.

The local schedule *sendl*, that is implemented as a bit-vector, indicates if the ABC is the sender in a given slot. Intuitively, in slot *s*, if $sendl[s] = 1$ then the ABC broadcasts the message stored in the send-buffer. Note that the ABC implementation is not aware of the round-number. It simply operates according to the slot-based fixed schedule, that is repeated time and again.

The special parameter *iwait* indicates the number of hardware cycles to be awaited before the ABC starts executing the schedule after power-up.

All parameters introduced so far are stored in configuration registers that need to be set by the host (we support memory mapped I/O) during an initialization phase. The host indicates that it has finished the initialization by invoking a *setrd* command. We do not go into details here, the interested reader may consult [7, 8].

Message Broadcast. The send-environment starts broadcasting the message contained in the send-buffer *sb* if the schedule-environment raises the *startsnd* signal.

The receive-environment permanently listens on the bus. At an incoming message, indicated by a falling edge (the bus is high-active), it signals the start of a reception to the schedule-environment by raising the *startedrcv* signal for one cycle. In addition it decodes the broadcast frame and writes the message into the receive buffer *rb*.

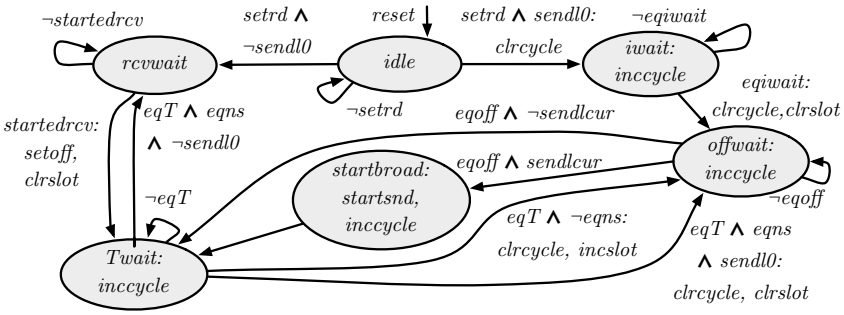

Fig. 1 Schedule Automaton

Scheduling. The schedule-environment maintains two counters: The cycle counter *cy* and the current slot counter *csn*. Both counters are periodically synchronized at the beginning of every round. All ECUs except the one broadcasting in slot 0 (we call the former *slaves* and the latter *master*) synchronize their counters to the incoming transmission in slot 0. Hence, the *startedrcv* signal from the receive environment is used to provide a synchronized time base (see below). Furthermore, the schedule-environment initiates the message broadcast by raising the *startsnd* signal for one cycle.

The schedule environment implements the automaton from Fig. 1. The automaton takes the following inputs: The *startedrcv* signal as described above. The signal *setrd* denotes the end of the configuration phase. The signal *sendl0* indicates if the ECU is the sender in the first slot and thus the master. Three signals are used to categorize the cycle counter; *eqiwait* indicates if the initial *iwait* cycles have been reached, similar to *eqoff* and *eqT*. The signal *eqns* indicates that the end of a round has been reached, i.e. that the slot counter equals $ns - 1$. Finally *sendlcur* indicates if the ABC is the sender in the current slot, i.e. $sendlcur = sendl[csn]$.

The automaton has six states and is clocked each cycle. Its functionality can be summarized as follows: If the *reset* signal is raised (which is assumed to happen only at power-up) the automaton is forced into the *idle*-state. If the host has finished the initialization and thus invoked *setrd* we split cases depending on the *sendl0* signal. If the ABC is the master, i.e. if *sendl0* holds, the ABC waits first *iwait* hardware cycles (in the *iwait*-state), then an additional *off* cycles (in the *offwait*-state) before it starts broadcasting the message (in the *startbroad*-state) and proceeds to the *Twait*-state.

If the ABC is a slave ($sendl0 = 0$), it waits in the *rcvwait*-state for an active *startedrcv* signal and then proceeds to the *Twait*-state. There all ABCs await the end of a slot indicated by *eqT*. Then we split cases if the round is finished or not. If the round is not finished yet (indicated by ¬*eqns*), all ABCs proceed to the *offwait*-state. Furthermore, the sender in the current slot (indicated by *sendlcur*) proceeds to the *startbroad*-state, initiates the message broadcast and then proceeds to the *Twait*-state; all other ABCs skip the *startbroad*-state and proceed directly to the *Twait*-state. At the end of a round, the master simply repeats the 'normal' sender

cycle (from the *Twait*-state to the *offwait*-state and finally to the *Twait*-state again). All other ABCs proceed to the *rcvwait*-state to await an incoming transmission.

Once initialized, the master ABC follows the schedule without any synchronization. At the beginning of a round it waits *off* many cycles and initiates the broadcast.

The clock synchronization on the slave ABCs is done in the *rcvwait*-state. In this state the cycle counter is not altered but simply stalls in its last value. At an incoming transmission (from the master) the slaves clear their slot-counter and set their cycle counter to *off*, i.e. the number of hardware cycles at which the master initiated the broadcast. After this all ABCs are (relatively) synchronized to the masters clock.

Hardware Construction. The number of ECUs connected to the bus is denoted ne. Thus an ECU number is given by $u \in [0 : ne-1]$. We use subscript ECU numbers to refer to single ECUs.

We denote the hardware configurations of ECU_u by h_u. If the index u of the ECU does not matter, we drop it. The hardware configuration is split into a host configuration and an ABC configuration. Since we do not go into details regarding the host, we stick to h to denote the configuration of our ABC. Its essential components are:

- Two single bit-registers, one for sending and one for receiving. Both are directly connected to the bus. We denote them $h.S$ and $h.R$.
- A second receiver register, denoted $h.\hat{R}$, to deal with metastability (see Sect. 3).
- Send buffer $h.sb$ and receive buffer $h.rb$ each capable of storing one message.
- The current slot counter $h.csn$ and the cycle counter $h.cy$.
- The schedule automaton is implemented straight-forward as a transition system on an unary coded bit-vector. We use $h.state$ to code the current state (see Fig. 1).
- Configuration registers.

The configuration registers are written immediately after reset / power-up. They contain in particular the locally relevant portions of the scheduling function.

To simplify arguments regarding the schedule we define a global scheduling function *send*. Given a slot-number s it returns the number of the ECU sending in this slot. Let $sendl_u$ denote the local schedule of ECU_u, then $send(s) = u \Leftrightarrow sendl_u[s] = 1$. Note that this definition implicitly requires a unique sender definition for each slot. Otherwise correct message broadcast becomes impossible due to bus contention.

Thus if ECU_u is (locally) in a slot with slot index s and $send(s) = u$ then ECU_u will transmit the content of the send buffer $h.sb$ via the bus during some transmission interval. A serial interface that is not actively transmitting during slot (r,s) puts by construction the idle value (the bit 1) on the bus.

If we can guarantee that during the transmission interval *all* ECUs are locally in slot (r,s), then transmission will be successful. The clock synchronization algorithm together with an appropriate choice of the transmission interval will ensure that.

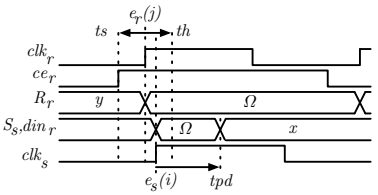

Fig. 2 Clock Edges

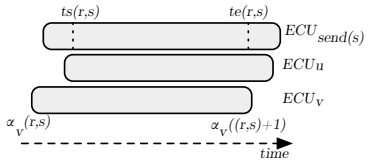

Fig. 3 Schedule

3 Verification

To argue about asynchronous distributed communication systems we have to formalize the behavior of the digital circuits connected to the analog bus. Using the formalization of digital clocks we introduce a hardware model for continuous time. In the remainder of this section we sketch the message transmission correctness, detail the scheduling correctness and combine both into a single correctness statement.

Clocks. The hardware of each ECU is clocked by an oscillator having a nominal clock period of τ_{ref}. The individual clock period τ_u of an ECU_u is allowed to deviate by at most $\delta = 0.15\%$ from τ_{ref}, i.e. $\forall u.\ |\ \tau_u - \tau_{ref}\ | \leq \tau_{ref} \cdot \delta$. Note that this limitation can be easily achieved by current technology.

Thus the relative deviation of two individual clock periods compared to a third clock period is bounded by $|\ \tau_u - \tau_v\ | \leq \tau_w \cdot \Delta$ where $\Delta = 2\delta/(1-\delta)$.

Given some clock-start offset $o_u < \tau_u$ the date of the clock edge $e_u(i)$ that starts cycle i on ECU_u is defined by $e_u(i) = o_u + i \cdot \tau_u$.

In our scenario all ECUs are connected to a bus. The sending ECUs broadcasts data which is sampled by all other ECUs. Due to clock drift it is not guaranteed, that the timing parameter of the sampling registers are obeyed. This problem is solved by serial interfaces. To argue formally we first introduce a continuous time model for bits being broadcast.

Hardware Model with Continuous Time. The problems solved by serial interfaces can by their very nature not be treated in a standard digital hardware model with a single digital clock clk. Nevertheless, we can describe each ECU_u in such a model having its own hardware configuration h_u.

To argue about the sender register $h.S$ of a sending ECU transmitting data via the bus to a receiver register $h.R$ of a receiving ECU, we have to extend the digital model.

For the registers connected to the bus –and only for those– we extend the hardware model such that we can deal with the concepts of propagation delay (tpd), setup time (ts), hold time (th), and metastability of registers. In the extended model used near the bus we therefore consider time to be a real valued variable t.

Next we define in the continuous time model the output of the sender register $h_u.S$ during cycle i of ECU_u, i.e. for $t \in (e_u(i) : e_u(i+1)]$. The content of $h_u.S$ at time t is

denoted by $S_u(t)$. In the digital hardware model we denote the value of some register, e.g. R, during cycle i by $h^i.R$ which equals the value at the clock edge $e_u(i+1)$.

If in cycle $i-1$ the digital clock enable $Sce(h_u^{i-1})$ signal was off, we see during the whole cycle the old digital value $h_u^{i-1}.S$ of the register. If the register was clocked ($Sce(h_u^{i-1}) = 1$) and the propagation delay tpd has passed, we see the new digital value of the register, which equals the digital input $Sdin(h_u^{i-1})$ during the previous cycle (see Fig. 2). Otherwise we cannot predict what we see, which we denote by Ω:

$$S_u(t) = \begin{cases} h_u^{i-1}.S & : Sce(h_u^{i-1}) = 0 \wedge t \in (e_u(i) : e_u(i+1)] \\ Sdin(h_u^{i-1}) & : Sce(h_u^{i-1}) = 1 \wedge t \in [e_u(i)+tpd : e_u(i+1)] \\ \Omega & : \text{otherwise} \end{cases}$$

The bus is an open collector bus modeled as the conjunction over all registers $S_u(t)$ for all t and u.

Now consider the receiver register $h_v.R$ on any ECU_v. It is continuously turned on; thus the register always samples from the bus. In order to define the new digital value $h_v^j.R$ of register R during cycle j on ECU_v we have to consider the value of the bus in the time interval $(e_v(j)-ts, e_v(j)+th)$. If during that time the bus has a constant digital value x, the register samples that value, i.e. $\exists x \in \{0,1\}.\ \forall t \in (e_v(j)-ts, e_v(j)+th).\ bus(t) = x \Rightarrow h_v^j.R = x$. Otherwise we define $h_v^j.R = \Omega$.

We have to argue how to deal with unknown values Ω as input to digital hardware. We will use the output of register $h_u.R$ only as input to a second register $h_u.\hat{R}$ whose clock enable is always turned on, too. If Ω is clocked into $h_u.\hat{R}$ we assume that $h_u.\hat{R}$ has an unknown but digital value, i.e. $h_u^j.R = \Omega \Rightarrow h_u^{j+1}.\hat{R} \in \{0,1\}$.

In real systems the counterpart of register $\hat{R}$ exists. The probability that R becomes metastable for an entire cycle *and* that this causes $\hat{R}$ to become metastable too is for practical purposes zero.

Continuous Time Lemmata for the Bus. Consider ECU_s is the sender and ECU_r is a receiver in a given slot. Let i be a sender *cycle* such that $Sce(h_s^{i-1}) = 1$, i.e. the output of S is not guaranteed to stay constant at time $e_s(i)$. This change can only affect the value of register R of ECU_r in cycle j if it occurs before the sampling edge $e_r(j)$ plus the hold time th, i.e. $e_s(i) < e_r(j)+th$. The first cycle that is possibly being affected is denoted by $cy_{r,s}(i) = \min\{j \mid e_s(i) < e_r(j)+th\}$.

In what follows we assume that all ECUs other than the sender unit ECU_s put the value 1 on the bus and keep their Sce signal off (hence $bus(t) = S_s(t)$ for all t under consideration). Furthermore, we consider only one receiving unit ECU_r. Because the indices r and s are fixed we simply write $cy(i)$ instead of $cy_{r,s}(i)$.

Theorem 1 (Message Broadcast Correctness). *Let the broadcast start in sender-cycle i. The value of the send buffer of $ECU_{send(s)}$ is copied to all receive buffers on the network side within tc sender cycles, i.e. $\forall u.\ h_u^{cy(i+tc)}.rb = h_{send(s)}^i.sb$.*

This theorem is proven by an in-depth analysis of the send-environment and the receive-environment. For details see [8]. We do not go into details regarding the message transmission here. Instead we focus on the scheduling correctness.

Scheduling. We assume w.l.o.g. that the ECU with number 0 is the master, i.e. $send(0) = 0$. Let p_u be the point in time when ECU_u is switched on. We assume that at most cp_{max} hardware cycles have passed on the master ECU from the point in time it was switched on until all other ECUs are switched on, too. Thus $\forall u. \mid p_u - p_0 \mid \leq cp_{max} \cdot \tau_0$.

Once initialization is done, all hosts invoke a *setrd* command. The master ECU waits *iwait* hardware cycles before it starts executing the schedule. We assume that that there exists a point in time denoted I_{max} at which all slaves have invoked the *setrd* command and await the first incoming message. This assumption can be easily discharged by deriving an upper bound for the duration of the initialization phase, say i_{max} hardware cycles in terms of the master ECU, and choosing *iwait* to be $cp_{max} + i_{max}$. The upper bound can be obtained by industrial worst case execution time (WCET) analyzers [6] for the concrete processor and software.

We introduce some notation to simplify the arguments regarding single slots. The start time of slot (r,s) on an ECU_u is denoted by $\alpha_u(r,s)$. Initially, for all u we define $\alpha_u(0,0) = I_{max}$. To define the slot start times greater than slot $(0,0)$ we need a predicate *schedexec* that indicates if the schedule automaton is in one of three *executing* states, i.e. $schedexec(h_u^i) = h_u^i.state \in \{offwait, Twait, startbroad\}$. Let c be the smallest local hardware cycle such that $e_u(c)$ is greater than $\alpha_u((r,s)-1)$, $schedexec(h_u^c)$ holds, $h_u^c.cy = 0$, and $h_u^c.csn = s$. Moreover let c' be the smallest cycle sucht that $e_u(c')$ is greater than $\alpha_u((r,s)-1)$ and $h_u^{c'}.state = rcvwait$.

$$\alpha_u(r,s) = \begin{cases} e_u(c) & : u = 0 \vee s > 0 \\ e_u(c') & : \text{otherwise} \end{cases}$$

Using the definition of a clock edge we obtain the hardware cycle corresponding to $\alpha_u(r,s)$, denoted by $\alpha t_u(r,s)$.

The local timers are synchronized each round. Next we define the point in time when the synchronization is done in round r. The synchronization end time of round r on ECU_u, denoted by $\beta_u(r)$, is defined similar to the slot start time. Let c be the smallest hardware cycle such that that $schedexec(h_u^c)$ holds, $cycle_u^c = off$, and $slot_u^c = 0$. Then $\beta_u(r)$ is defined by $e_u(c)$.

Lemma 1 (Synchronization Times Relation). *For all u the synchronization of ECU_u to the master is completed within the adjustment time $ad = 10$ cycles relative to an arbitrary clock period τ_w, i.e. $\beta_0(r) = \alpha_0(r,0) + off \cdot \tau_0$ and $\beta_u(r) < \beta_0(r) + 10 \cdot \tau_w$*

The proof of this lemma is split in two parts. First, an analysis of the sender bounds the delay between an active *startsnd* signal and the actual transmission start. Second, we need to bound the delay on the receiver side until the *startedrcv* signal is raised after an incoming transmission plus an additional cycle to update the counters and the schedule control automaton. Next we relate the start times of slots on the same ECU.

Lemma 2 (Slot Start Times Relation). *The start of slot (r,s) on the master ECU depends only on the progress of the local counter, i.e. $\alpha_0(r,s) = \alpha_0((r,s)-1) + T \cdot \tau_0$. The start of slot (r,s) on all other ECUs is given by:*

$$\alpha_u(r,s) = \begin{cases} \beta_u(r) + (T - off) \cdot \tau_u & : s = 1 \\ \alpha_u((r,s)-1) + T \cdot \tau_u & : s \neq 1 \end{cases}$$

Proof by induction on r and s using arguments for the concrete hardware.

The transmission is started in slot (r,s) by $ECU_{send(s)}$ if the local cycle count equals *off*. This point in time is denoted by $ts(r,s) = \alpha_{send(s)}(r,s) + off \cdot \tau_{send(s)}$. According to Theorem 1 the transmission ends at time $te(r,s) = ts(r,s) + tc \cdot \tau_{send(s)} = \alpha_{send(s)}(r,s) + (off + tc) \cdot \tau_{send(s)}$.

The schedule is correct if the transmission interval $[ts(r,s), te(r,s)]$ is contained in the time interval, when all ECUs are in slot (r,s), as depicted in Fig. 3.

Theorem 2 (Schedule Correctness). *All ECUs are in slot (r,s) before the transmission starts. Furthermore, the transmission must be finished before any ECU thinks it is in the next slot, i.e. $\alpha_u(r,s) < ts(r,s)$ and $te(r,s) < \alpha_u((r,s)+1)$*

This theorem is proven by a case split on (r,s) using Lemmata 1 and 2. Now we can state the overall transmission correctness in the digital hardware model:

Theorem 3 (Overall Transmission Correctness). *Consider slot (r,s). The value of the send buffer of $ECU_{send(s)}$ at the start of slot (r,s) is copied to all receive buffers by the end of that slot, i.e. $\forall u.\ h_u^{\alpha t_u((r,s+1))-1}.rb = h_{send(s)}^{\alpha t_{send(s)}(r,s)}.sb$*

To prove this theorem we combined Theorem 1 and Theorem 2. According to Theorem 1 the actual broadcast is correct if the transmission window $[ts(r,s), te(r,s)]$ is big enough. The latter is proven by Theorem 2.

4 Conclusion

In this paper we present a formal correctness proof of a distributed automotive system at gate-level (Sect. 3) along with its hardware implementation (Sect. 2). The hardware model has been formalized in Isabelle/HOL on boolean gates.

While a simple version of the message transmission correctness has already been published before [8, 16], in this new work, we have formally analyzed the scheduler itself and have integrated both results into a single correctness statement. All lemmata relating to scheduling correctness have been formally proven in Isabelle/HOL which took about one person year.

We used automatic tools as the symbolic, open source model checker NuSMV, to discharge properties related to bit-vector operations and the schedule automaton of the hardware. With our implementation heavily using bit-vectors, we ran into the infamous state explosion problem. By resorting to IHaVeIt (a domain-reducing preprocessor for model checkers) we were able to cope with this problem. However, missing support for real-linear arithmetic in the automatic tool landscape, made the verification of the analog and timed models tedious. Yet the integration of decision procedures of dense-order logic would be helpful. In short: automatic tools took a

heavy burden from us in the digital world but were almost useless for continous-timed analysis.

Summing up, our work provides a strong argument for the feasibility of formal and pervasive verification of concrete hardware implementations at gate-level.

References

1. Berry, G., Kishinevsky, M., Singh, S.: System level design and verification using a synchronous language. In: ICCAD, pp. 433–440 (2003)
2. Bevier, W., Young, W.: The proof of correctness of a fault-tolerant circuit design. In: Second IFIP Conference on Dependable Computing For Critical Applications, pp. 107–114 (1991)
3. Beyer, S., Böhm, P., Gerke, M., Hillebrand, M., In der Rieden, T., Knapp, S., Leinenbach, D., Paul, W.J.: Towards the formal verification of lower system layers in automotive systems. In: ICCD '05, pp. 317–324. IEEE Computer Society (2005)
4. Brown, G.M., Pike, L.: Easy parameterized verification of biphase mark and 8N1 protocols. In: TACAS'06, *LNCS*, vol. 3920, pp. 58–72. Springer (2006)
5. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Marco Pistore, M.R., Sebastiani, R., Tacchella, A.: NuSMV 2: An open source tool for symbolic model checking. In: CAV '02, pp. 359–364. Springer-Verlag (2002)
6. Ferdinand, C., Martin, F., Wilhelm, R., Alt, M.: Cache Behavior Prediction by Abstract Interpretation. Sci. Comput. Program. **35**(2), 163–189 (1999)
7. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD '05, pp. 309–316. IEEE Computer Society (2005)
8. Knapp, S., Paul, W.: Realistic Worst Case Execution Time Analysis in the Context of Pervasive System Verification. In: Program Analysis and Compilation, *LNCS*, vol. 4444, pp. 53–81 (2007)
9. Lamport, L., Melliar-Smith, P.M.: Synchronizing clocks in the presence of faults. J. ACM **32**(1), 52–78 (1985)
10. Miner, P.S., Johnson, S.D.: Verification of an optimized fault-tolerant clock synchronization circuit. In: Designing Correct Circuits. Springer (1996)
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
12. Pfeifer, H., Schwier, D., von Henke, F.W.: Formal verification for time-triggered clock synchronization. In: DCCA-7, vol. 12, pp. 207–226. IEEE Computer Society, San Jose, CA (1999)
13. Pike, L.: Modeling Time-Triggered Protocols and Verifying Their Real-Time Schedules. In: FMCAD'07, pp. 231–238 (2007)
14. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering **25**(5), 651–660 (1999)
15. Rushby, J.: An overview of formal verification for the time-triggered architecture. In: FTRTFT'02, *LNCS*, vol. 2469, pp. 83–105. Springer-Verlag, Oldenburg, Germany (2002)
16. Schmaltz, J.: A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware. In: FMCAD'07, pp. 223–230. IEEE/ACM, Austin, TX, USA (2007)
17. Shankar, N.: Mechanical verification of a generalized protocol for byzantine fault tolerant clock synchronization. In: FTRTFT'92, vol. 571, pp. 217–236. Springer, Netherlands (1992)
18. Tverdyshev, S., Alkassar, E.: Efficient bit-level model reductions for automated hardware verification. In: TIME 2008, to appear. IEEE Computer Society Press (2008)
19. Zhang, B.: On the Formal Verification of the FlexRay Communication Protocol. Automatic Verification of Critical Systems (AVoCS'06) pp. 184–189 (2006)